Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Amine El Ayadi

Einsatz von SOA in E-Commerce Systemen

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Amine El Ayadi

Einsatz von SOA in E-Commerce Systemen

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang Master of Science Informatik

am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr.Ing. Wendholt Birgit
Zweitgutachter : Prof. Dr. rer. nat. Klemke Gunter

Abgegeben am 23. Dezember 2010

Amine El Ayadi

Thema der Masterarbeit

Einsatz von SOA in E-Commerce Systemen

Stichworte

SOA, ESB, Webservives, SOAP, JBOss, Intershop, Integration, Migration, Modularisierung, Elektronischer Handel

Kurzzusammenfassung

In dieser Arbeit wird ein Konzept für eine Architektur entwickelt, mit der mit Hilfe von SOA die Integrationsproblematik in E-Commerce-Systemen gelöst werden kann. Es werden Integrationsansätze untersucht, die das Reengineering bestehender Software vereinfachen können. Dazu werden die Möglichkeiten untersucht, wie sich Logik aus dem bestehenden System am besten extrahieren lässt. Als Ergebnis dieser Arbeit wurde ein Designkonzept entwickelt, das ermöglicht, beliebig erweiterbare Softwarekomponenten zu entwickeln, die vielseitig in E-Commerce-Systemen eingesetzt werden können

Amine El Ayadi

Title of the paper

Applying SOA to an Ecommerce system

Keywords

SOA, ESB, Webservice, JBoss, SOAP, integration, Intershop, migration, Reengineering, E-Commerce

Abstract

In this paper a software design is developed, which can solve the integration issues in e-commerce systems by using SOA. Multiple integration approaches for simplification of re-engineering of existing software are examined to extract logic from existing services. The result of this paper is a design concept that supports the development of arbitrary extensible software components for universal usage in e-commerce systems.

Inhaltsverzeichnis

Abbildungsverzeichnis **6**

1 Einführung und Motivation **8**
1.1 **E-Commerce** . . . 8
1.2 **Motivation** . . . 9
1.2.1 **Die Rolle für SOA für Softwareintegration und Migration** . . . 14
1.3 **Zielsetzung** . . . 15
1.4 **Gliederung der Arbeit** . . . 17

2 Technische Grundlagen **19**
2.1 **SOA** . . . 20
2.1.1 **Kernidee einer SOA** . . . 21
2.1.2 **SOA Schichten** . . . 22
2.1.3 **Zusammenfassung** . . . 26
2.2 **Webservices** . . . 27
2.2.1 **SOAP-Webservices** . . . 27
2.3 **Zusammenfassung** . . . 28

3 Vergleichbare Arbeiten **30**
3.1 **Ansätze aus der Literatur** . . . 31
3.1.1 **Integrationsansätze** . . . 31
3.1.2 **Migrationsprozesse** . . . 36
3.2 **Ansätze aus dem Praxis** . . . 45
3.2.1 **SOA-Einsatz bei T-Com** . . . 46
3.2.2 **SOA-Einsatz bei Deutsche Post Brief** . . . 49
3.3 **Zusammenfassung** . . . 55

4 Analyse **57**
4.1 **Anwendungs-Szenario** . . . 57
4.2 **Analyse des Ist-Zustandes** . . . 60
4.2.1 **Softwarearchitektur** . . . 60
4.2.2 **Systemarchitektur** . . . 62
4.2.3 **Prozess Beschreibung** . . . 65

4.2.4 **Zusammenfassung** . . . 68
4.3 **Anforderungsanalyse** . . . 69
4.3.1 **Motivation zur Lösung der Integrationsproblematik** . . . 69
4.3.2 **Funktionale Anfoderungen** . . . 70
4.3.3 **Nichtfunktionale Anforderungen** . . . 76
4.3.4 **Zusammenfassung** . . . 79

5 Design **80**
5.1 **System-Architektur** . . . 80
5.1.1 **Zielarchitektur** . . . 82
5.1.2 **Integrationsaufgabe** . . . 84
5.2 **Entwurf** . . . 86
5.2.1 **JBoss ESB** . . . 86
5.2.2 Komponenten-Modell . . . 96
5.2.3 **Servicemodellierung** . . . 97
5.2.4 **Zusammenfassung** . . . 103

6 Prototypische Evaluierung **104**
6.1 **Entwicklungsumgebung und benötigte Tools** . . . 104
6.1.1 **Entwicklungsplattform (Enfinity Studio)** . . . 104
6.1.2 **JBoss ESB Applikationsserver** . . . 105
6.2 **Realisierung** . . . 105
6.2.1 **JBoss Aktion Implementierung** . . . 106
6.2.2 **Serviceimplementierung** . . . 107
6.2.3 **Client Anwendung** . . . 108

7 Fazit **110**
7.1 **Zusammenfassung der Ergebnisse** . . . 110
7.2 **Ausblick** . . . 113

Literaturverzeichnis **115**

Abbildungsverzeichnis

1.1 Rasantes Wachstum im E-Commerce Bundesverband Informationswirtschaft (2010) . . . 9
1.2 Aktuelle Architektur des E-Commerce Unternehmens . . . 11
1.3 Geschäftsprozesse der Beispiel-Unternehmen . . . 13
1.4 Ziel Architektur . . . 16

2.1 Die drei Rollen der Webservices . . . 21
2.2 Die Bestandteile des SOA-Modells . . . 23
2.3 Web Services-Oriented Architektur . . . 26
2.4 Webservice Architektur . . . 28

3.1 Schichten des Altsystems . . . 38
3.2 Beispiel eines Gateways . . . 39
3.3 Schritte des Chicken-Little-Ansatzes . . . 40
3.4 Butterfly-Ansatz: In Anlehnung an Wu et al. (1999), S. 3. . . . 42
3.5 Servicebeschreibung im zentralen Repository (Quelle: T-Com) . . . 47
3.6 Elemente der SOA-Integrationsinfrastruktur . . . 48
3.7 Applikationsdomänen und Zugriff über Services am Beispiel des Prozesses . . . 50
3.8 Implementierung des Services Kundenmanagement . . . 52
3.9 Komponenten des SBB (in Weiterentwicklung) [SANNEMANN04] . . . 53

4.1 Ablaufdiagram CallAgent Bestellung . . . 59
4.2 Softwarerchitektur des Unternehmens . . . 61
4.3 Systemarchitektur des Unternehmens . . . 62
4.4 Prozessablauf Produkte . . . 65
4.5 Prozessablauf Bestellung . . . 66
4.6 Prozessablauf Retoure . . . 66
4.7 Prozessablauf Stornierung . . . 67
4.8 Geschäftsablauf Bestellung erzeugen . . . 68
4.9 Anwendungsfälle aus Nutzersicht . . . 71
4.10 Shop-System Komponente . . . 74

5.1 Systemarchitektur des Altsystems . . . 81

5.2 Architektur des Zielsystem 83
5.3 Services im JBoss 87
5.4 Messages Aufbau im JBoss 89
5.5 SOAP Processor Aktion 93
5.6 SOAP ClientAction 94
5.7 Komponenten Diagramm 96
5.8 Server-Anwendung der AddressValidator 98
5.9 Client-Anwendung des AddressValidator 102

1 Einführung und Motivation

1.1 E-Commerce

Elektronischer Handel (engl. E-Commerce), der Handel über elektronische Kommunikationswege, ist ein Konzept, das seit dem Aufkommen elektronischer Netzwerke untersucht wurde T. Malone (2009). Mit der explosionsartigen Verbreitung des Internet Ende der 90er Jahre ist dabei immer mehr der Handel über das Internet, sowohl im Privat- als auch im Geschäftsbereich, in das Zentrum des Interesses gerückt. Obwohl die enormen Wachstumsraten, die Analysten dem Geschäft über das Internet vorausgesagt haben, nicht eingetreten sind, hat sich der elektronische Handel etabliert und ist heute fester Bestandteil der Geschäftswelt.

E-Commerce Lösungen werden hauptsächlich in Bereichen eingesetzt, in denen gut beschreibbare Güter in größeren Stückzahlen gehandelt werden. Um den Handel von hochwertigen und komplexen Produkten zu unterstützen, werden in der Forschung Marktplätze betrachtet, die ein Forum für das Zusammenbringen von vielen Anbietern und Nachfragern schaffen Schoop (2003). Solche Marktplätze unterscheiden sich von den zur Zeit eingesetzten Lösungen dadurch, dass nicht Handel mit dem Marktplatzbetreiber, als Anbieter oder Nachfrager, getrieben wird, sondern der Marktplatz eine Vermittlerrolle zwischen Marktplatzteilnehmern annimmt.

Speziell beim E-Commerce nimmt Deutschland weiterhin eine Spitzenposition in Europa ein. Das zeigen nicht nur die Studienergebnisse, sondern auch Zahlen, die der Bundesverband Informationswirtschaft, Telekommunikation und neue Medien (BITKOM) erhoben hat. Der deutsche Online-Handel wird weiterhin stark wachsen. Demnach werden für 2010 E-Commerce-Umsätze in Höhe von 781 Milliarden Euro prognostiziert. 2006 wurden rund 30 Prozent aller in Westeuropa über das Internet gehandelten Waren und Dienstleistungen hierzulande verkauft. Der gesamte Umsatz im deutschen Online-Handel stieg 2006 im Vergleich zum Vorjahr um 58 Prozent auf 321 Milliarden Euro (s. Abbildung 1.1). Auch in den kommenden Jahren wird sich das Wachstum fortsetzen.

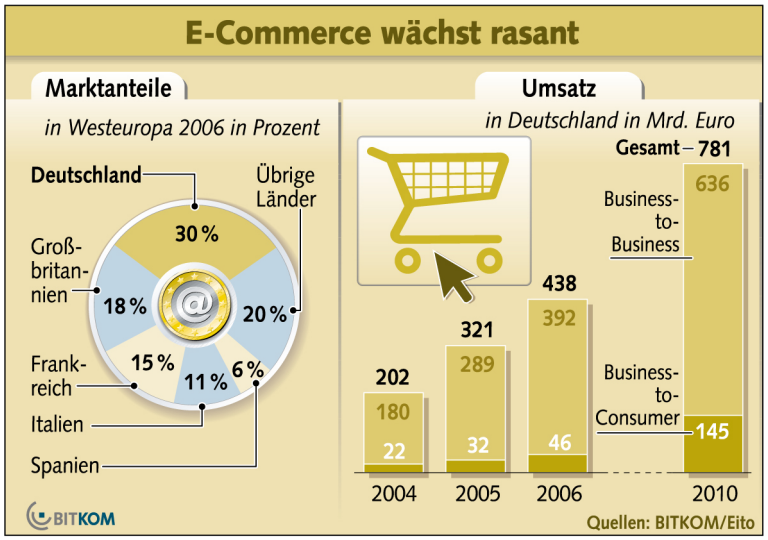

Abbildung 1.1: Rasantes Wachstum im E-Commerce Bundesverband Informationswirtschaft (2010)

1.2 Motivation

Mit dem rasanten Wachstum des E-Commerce entstehen immer neue Anforderungen und höhere Ansprüche an IT-Systeme, wodurch auch Kosten und Komplexität der IT-Projekte wachsen. Doch gründliche Analyse und genauere Planung werden sehr oft aus organisatorischen Gründen vernachlässigt. Dies führt dazu, dass IT-Projekte scheitern oder Kosten und Zeit nicht mehr zu kontrollieren sind, besonders, wenn keine vollständige, bzw. stabile ***Architektur*** geplant wurde. Somit werden IT-Architekturen in Unternehmen, gerade für eine erfolgreiche IT-Governance, umso wichtiger und gewinnen zunehmend Bedeutung.

Es existiert kein Unternehmen mehr, das keine Software einsetzt. Im allgemeinen gilt die Faustformel: Je größer das Unternehmen, desto größer sind die Softwaresysteme. Diese großen Systeme sind meistens sehr komplex oder sie sind eine Mischung aus Software und menschlichen Aktionen. Als Lösung dieser Komplexität kommt meistens die Idee, die Systeme zu strukturieren, indem Services in die IT-Architektur eingeführt werden. Diese Strukturierung soll demnächst eine Automatisierung, bzw. eine Erleichterung der Geschäftsabläufe ermöglichen, damit die Integration neuer Applikationen mit weniger Komplexität durchgeführt werden kann. Schon bei der Einführung oder Strukturierung einer IT-Architektur sollen die richtigen Werkzeuge und Methoden bestimmt werden.

Das Angebot an Methoden und Möglichkeiten zur Konzeption, Implementierung und Überwa-

chen von **IT-Architekturen** wird größer und oft verwirrend. Dies führt dazu, dass sich It- und Fach- Abteilungen nicht verständigen können und meistens aneinander vorbei arbeiten.

Der Architekturbegriff wird meistens in den folgenden unterschiedlichen Bereichen verwendet:

- **Informationsarchitektur** definiert die Beziehungen zwischen verschiedener Informationssystemen oder Teilen eines Informationssystems.
- **Softwarearchitekturen** stellen die Zusammenhänge zwischen Elementen von Softwaresystemen dar.
- **Rechnerarchitekturen** analog zu den Softwarearchitekturen beschreiben die Rechnerarchitekturen die Zusammenhängen und Beziehungen zwischen den Komponenten auf der Hardware-Ebene.

Die drei vorgestellten Architekturen schaffen zusammen eine „Grundlage“ für die Konstruktion einer IT-Systemarchitektur. Sie dient als Basis für die Entwicklung, Anpassung, Erweiterung und Wartung der Infrastruktur einer Organisation und ist damit der Schlüssel für ein hochqualifiziertes und erfolgreiches Informationsmanagement.

Bei der Planung einer neuen Architektur werden häufig hohe Anforderungen gestellt. Die Kosten und Komplexität des Systems sollen möglichst gering sein. Dazu soll die Architektur flexibel und einfach erweiterbar sein, mit der Möglichkeit, (heterogene-) Legacy-Systeme zu behalten und wieder zu verwenden. Die Wiederverwendung von standardisierten Softwarekomponenten kann hierbei die Kosten des Systems verkleinern. Schon auf der ersten und legendären Software-Engineering-Konferenz 1968 in Garmisch-Partenkirchen wurde die Idee vorgestellt, Software in Zukunft nicht mehr durch vollständige Neuentwicklungen, sondern durch das Wiederverwenden, bzw. Zusammenstecken von standardisierten Softwarekomponenten zu erstellen. Dort wurde die Idee geboren, service-orientierte Dienste anzubieten, indem Services zur Verfügung gestellt werden, die betriebsintern und bei Bedarf betriebsübergreifend verwendet werden können und bei einem Technologiewechsel weiterhin bestehen bleiben. Die Realisierung von Softwarearchitekturen auf der Basis von service-orientierten Diensten, inzwischen Service Oriented Architecture (SOA) genannt, bietet nicht nur die Möglichkeit, standardisierte Softwarekomponenten zu verwenden, sondern auch eine ausfallsichere Architektur. Ein Service im Kontext einer SOA sollte die Fachlichkeit unterstreichen und keinerlei technologische Information beinhalten. Der Service kann betriebsintern sowie von Geschäftspartnern verwendet werden, auch wenn sie unterschiedliche Plattformen verwenden.

Das Ziel einer SOA ist es, Services bereitzustellen, welche von vielen Interessenten benutzt werden können, ganz unabhängig davon, welche Technologie dahinter steckt. Dadurch werden die einzelnen Prozesse umgelagert mit der Gewährleistung, erfolgreich und in der richtigen Reihenfolge abgearbeitet zu werden. Somit spielt die Interoperabilität bei einer SOA eine

wichtige Rolle. Um diese sicherzustellen, sind bestimmte Voraussetzungen, wie beispielsweise Standardisierung, zu erfüllen.

Diese Arbeit basiert auf der Architektur eines E-Commerce-Unternehmens. Abbildung 1.2 beschreibt diese Architektur:

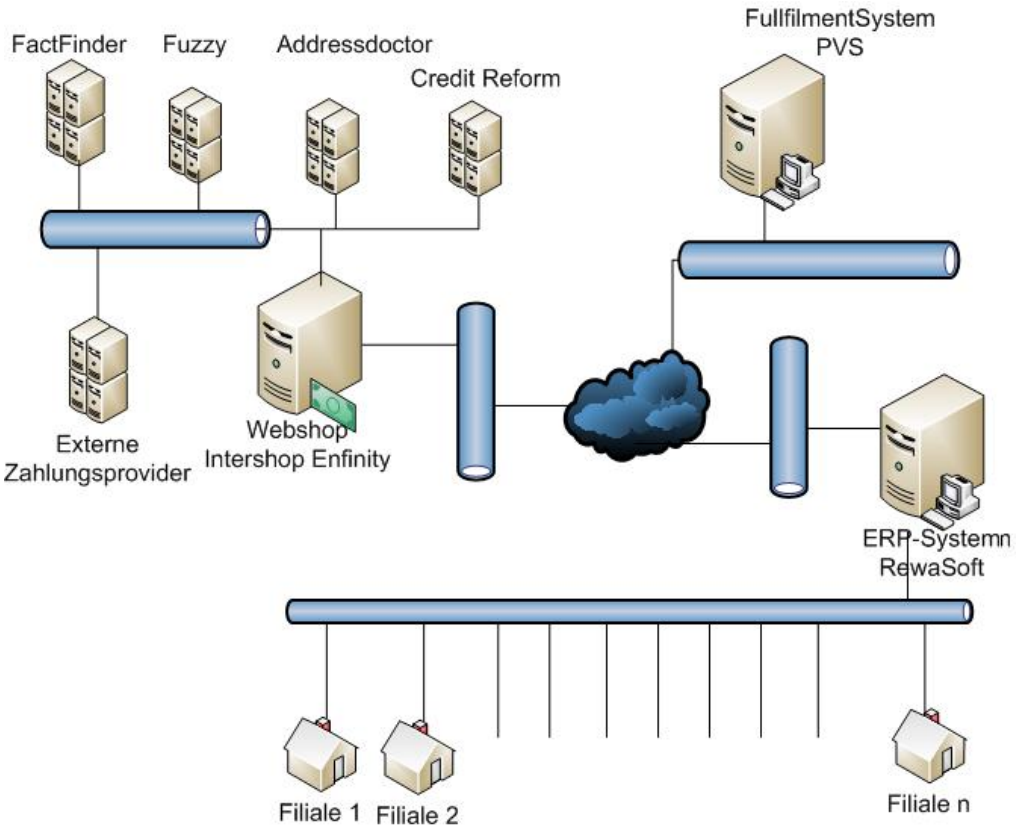

Abbildung 1.2: Aktuelle Architektur des E-Commerce Unternehmens

Im Mittelpunkt dieser Architektur steht das Shop-System auf der Basis einer Standard-Lösung von Intershop Enfinity. Dabei ruft das Shop-System externe Dienste auf, die unterschiedliche Funktionalitäten anbieten:

- **Adressenvalidierung**

  Um Kunden-Adressen zu validieren, werden externe Dienste verwendet (Fuzzy und AdressDoctor).

- **Produktsuche** (FactFinder)

Bei der Produktsuche wird der Factfinder-Server eingesetzt. Dieser Server bietet einen Service für die Produktsuche, der über Webservices aufgerufen wird.

- **Zahlung**

  Um Zahlungen durchzuführen, werden unterschiedliche Zahlungsmethoden angeboten. Dabei werden die Zahlungen meistens über externe Dienste abgewickelt (z. B. Paypal, Sofortüberweisung oder Saferpay)

- **Fullfilment-System:** Das Fullfilment-System ist dafür zuständig, die Logistik zu übernehmen. Dabei werden alle Produkt- und Bestellinformationen mit dem Shop-System über bestehende Services ausgetauscht.

ERP-System Auch das ERP-System tauscht die Bestellungs und Produkt- Informationen mit dem Shop-System aus. Dabei bietet das ERP-System mehrere Funktionalitäten an:

- Stornoservice
- Bestellservice
- Retoure-Importservice
- Retoure-Exportservice
- Produktservice

Das E-Commerce-Unternehmen verwendet eine eigene Prozess-Engine, um die auszuführenden Prozesse - also die vorstrukturierte Abfolge von einzelnen Aktivitäten - zu steuern. Momentan werden folgende Prozesse im E-Commerce-Unternehmen (Abbildung 1.3) abgebildet:

Die Abbildung beschreibt die Abfolge der folgenden Prozesse:

- **Prozess-Ablauf-Produkte:**
- **Prozess-Ablauf-Bestellungen:**
- **Prozess-Ablauf-Retoure:**
- **Prozess-Ablauf-Stornierung:**

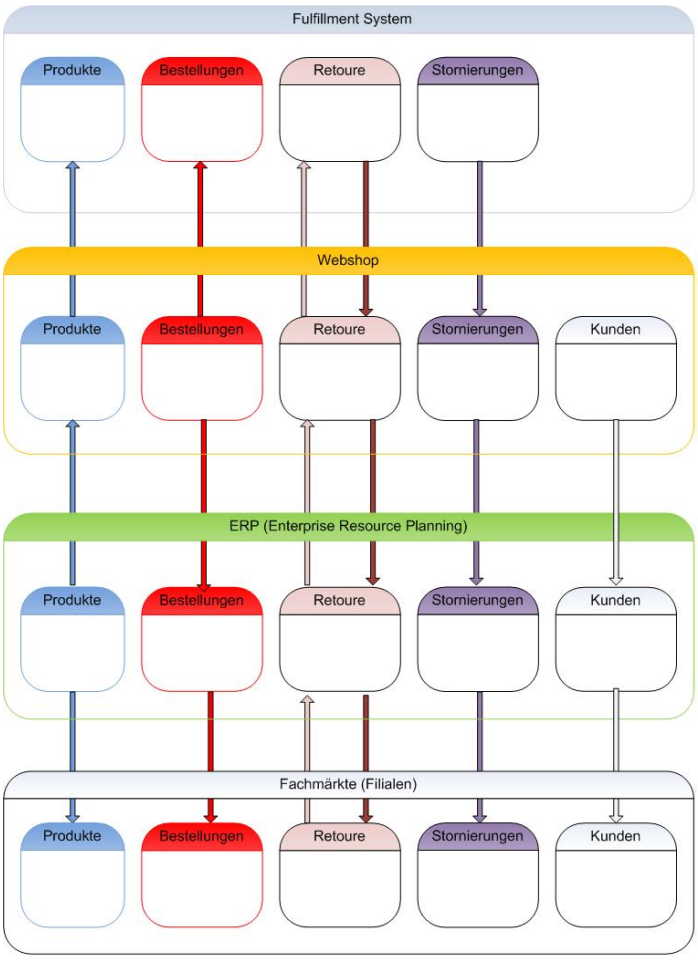

Abbildung 1.3: Geschäftsprozesse der Beispiel-Unternehmen

Im Kapitel „Analyse“ 4.2 werden die dargestellten Prozesse einzeln beschrieben.

Die Architektur des Systems ist stabil und erfüllt die Anforderungen des Unternehmens. Beim Versuch, das Shop-System mit einem zusätzlichen Vertriebs-Kanal zu erweitern ist das Problem entstanden, dass die Architektur nicht erweiterbar ist und die Funktionalitäten nicht modular sind. Genauer definiert hat das Unternehmen versucht, eine neue Applikation bzw. Komponente hinzuzufügen, um telefonische Bestellungen anzunehmen und zu generieren. Dabei sollen die gleichen Zahlungsmethoden und Adressen-Validierungs-Dienste verwendet werden bzw. neu implementiert werden. Das E-Commerce Unternehmen besitzt eine Software, die nicht einfach erweiterbar ist und mit diesem Problem konfrontiert ist. Es existieren Architektu-

ren, die sich anbieten, wenn eine Modularisierung vorgesehen ist. Damit wird es relativ einfach sein, diese Module in verschiedene Kanäle zu integrieren.

Speziell für E-Commerce-Systeme werden sehr oft solche Anforderungen nachträglich eingeführt. Mit der aktuellen Architektur ist diese Erweiterung möglich, aber nur mit extrem hohem Aufwand. Durch die integrierte Logik bei den einzelnen Funktionalitäten und Services, die feste Koppelung des ERP-Systems und Fullfilment-Systems mit dem Shop-System in die bestehende Architektur (Abbildung 1.2) entstand ein monolithisches Shop-System. Die Integration neuer Applikationen wird komplizierter und die Wartbarkeit des Systems aufwändiger. Dadurch werden auch die Entwicklungskosten mehrfach dupliziert und dazu entstehen zunehmend komplexere IT-Systeme. Die komplexen IT-Systeme stellen auch eine Herausforderung bei der Integration neuer Komponenten dar, die am Anfang nicht bekannt waren.

Die Beschäftigung mit der Integration verschiedener Anwendungen ist ein altbekanntes und stets aktuelles Problem für IT-Strategen. Durch den Einsatz von Service Oriented Architecture (SOA) wird versucht, allgemeine Integrationsprobleme zu vermeiden, bzw. zu lösen. Im nächsten Abschnitt wird im Rahmen einer Allgemeinen SOA-Definition Bezug genommen auf Lösungsansätze zur Erfüllung der genannten nicht funktionalen Anforderungen.

1.2.1 Die Rolle für SOA für Softwareintegration und Migration

SOA als Architektur ermöglicht es, schnell und einfach auf Veränderungen zu reagieren. Dazu bietet SOA eine einfache Integration von internen und externen Systemen. Die Wiederverwendung von den Komponenten wird vereinfacht und dadurch wird eine hohe Wartbarkeit und geringere Komplexität des Systems erreicht.

Unter dem Gesichtspunkt Zuverlässigkeit und Reduktion der Komplexität bietet SOA das Konzept des ESB. Ein ESB ist eine Middleware-Komponente, die auf ein Messaging-System aufsetzt und eine Infrastruktur anbietet, die eine lose Kopplung von verteilten Services ermöglicht. Die Zuverlässigkeit wird durch den Einsatz vom Enterprise Service Bus (ESB) ermöglicht. Dabei wird auch gewährleistet, dass die einzelnen Aufträge, bzw. Anfragen komplett und in der richtigen Reihenfolge abgearbeitet werden, da der ESB auch dafür ausgelegt ist, hohen und sicheren Datenaustausch für eine große Anzahl an Services und Anwendern zu garantieren. Für die Anwendungsintegration baut die SOA auf Standards auf. Die Benutzung von Standards sichert die technische Interoperabilität. Daher bezeichnet das ESB einen weiteren Aspekt der dafür spricht, SOA-Architektur in E-Commerce-Unternehmen einzusetzen.

Unter dem Gesichtspunkt „Erweiterbarkeit und Zuständigkeit“ wird im Rahmen dieser Arbeit unter Berücksichtigung des Architekturstils von SOA ein Designkonzept erstellt, das die Einführung von SOA erleichtert und das Ziel hat, Integrationsprobleme zu vermeiden, bzw. zu beseitigen. Im folgenden Abschnitt werden die Ziele dieser Arbeit detaillierter beschrieben.

1.3 Zielsetzung

Das Ziel dieser Arbeit ist, das Konzept für eine Architektur zu entwickeln, welche beschreibt, wie die Realisierung einer SOA hilft, die Integrationsproblematik in E-Commerce-Systemen zu lösen, bzw. zu vermeiden. Dabei wird die Integrationsproblematik speziell bei E-Commerce-Systemen abgegrenzt. Dann sollen die einzelnen Komponenten einer SOA-Architektur defniert werden mit der Untersuchung, inwiefern sie zur Lösung der Integrationsproblematik beitragen können.

Für die Integration neuer Komponenten werden Integrationsansätze untersucht, die das Re-Engineering bestehender Software vereinfachen können. Dabei werden die Möglichkeiten untersucht, wie sich Logik aus dem bestehenden System am besten extrahieren lässt.

Als „Proof of Concept“ werden zwei Validierungsdienste in einem neuen Modul integriert und mit einem ESB verbunden. Dazu wird die Clientanwendung entwickelt, die zwei unterschiedliche Adressen-Validierungs-Systeme über den ESB aufrufen kann. Bei der Integration der Validierungsdienste in den ESB soll ein Laufzeit-Modell entwickelt werden, welches die beiden Validierungsdienste hinter einer abstrakten Schnittstelle abstrahiert und somit eine Integration von zusätzlichen Diensten zur Laufzeit ermöglicht.

Abbildung 1.4 gibt einen Überblick über die neue Architektur.

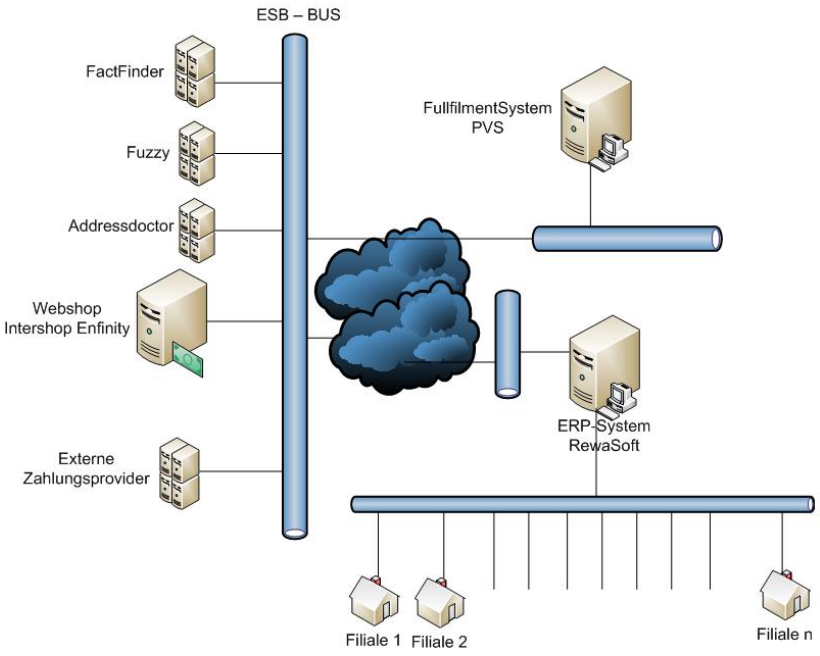

Abbildung 1.4: Ziel Architektur

In der neuen Architektur übernimmt der ESB die zentrale Rolle anstelle des Shop-Systems. Dabei werden alle Komponenten direkt mit dem ESB integriert, sodass sie mehrfach und von unterschiedlichen Applikationen verwendet werden können. Applikationen, die eine ähnliche Funktionalität anbieten, werden hinter generischen Schnittstellen abstrahiert.

Desweiteren besteht das Ziel dieser Arbeit auch darin, zu untersuchen, wie der Einsatz von SOA in E-Commerce-Systemen zu realisieren ist, um allgemeine Integrationsaufgaben zu lösen und welche Vorteile dadurch entstehen können. Dabei werden die möglichen Integrationsansätze vorgestellt und untersucht, welche davon für E-Commerce-Systeme am besten geeignet sind.

1.4 Gliederung der Arbeit

Zuerst werden die technischen Grundlagen definiert, die in dieser Arbeit benötigt werden (Kapitel 2). Dabei wird zuerst eine Zusammenfassung der Grundlagen von SOA geben, sowie die Schichten einer SOA definiert, um zu ermitteln, welche SOA-Hilfskomponente, bzw. Schichten am besten für E-Commerce Systemen geeignet sind. In diesem Kapitel wird eine Begründung geben, warum die einzelnen SOA-Schichten für das E-Commerce geeignet oder nicht geeignet sind.

Nach der Defnition der Grundlagen wird im dritten Kapitel dieser Arbeit aufgeführt, welche theoretischen oder praktischen Ansätze bereits zur Lösung von allgemeinen Integrationsaufgaben und die darauf aufbauende Migrationsproblematik existieren (Kapitel 3). Dabei werden Lösungen aus der Literatur(Kapitel 3.1) sowie aus der Praxis (Kapitel 3.2) vorgestellt. Es werden drei bekannte Integrationsansätze vorgestellt. Es wird gezeigt, dass für das Re-Engineering von Funktionalitäten, die externe Dienste aufrufen, das Wrapping Ansatz am besten geeignet ist. Für Funktionalitäten mit großem Anteil an E-Commerce-Logik wird das „Neuentwickeln von Teilkomponente“ als geeigneter Ansatz erklärt. Nach der Vorstellung der existierenden Integrationsansätze, werden die existierenden Migrationsprozesse erklärt. Dabei werden die einzelnen Schritte definiert, die für die Migration einer bestehenden Applikation in eine SOA-Architektur benötigt werden. Im Allgemeinen werden Migrationsansätze untersucht, die die Verfügbarkeit des Altsystems gewährleisten, während das neue System mit Daten versorgt wird. Danach findet eine Auswertung der vorgestellten Integration und Migrationsansätze statt. Es wird gezeigt, dass der Cold-Turkey-Ansatz am besten für E-Commerce Systeme geeignet ist.

Zuletzt werden zwei bekannte Praxisbeispiele (Deutsche Post und T-COM) einer SOA-Umsetzung vorgestellt (Kapitel 3.2).

Im Kapitel 4 (Analyse) wird zuerst ein mögliches Anwendungs-Szenario beschrieben. Daraus werden die Anforderungen für die neue Architektur extrahiert. Hiernach wird der Umfang der Zielarchitektur festgelegt. Desweiteren wird die aktuelle Architektur des E-Commerce Unternehmens vorgestellt und analysiert. Dabei werden die Ziele dieser Arbeit überführt in eine präzise Aufgabenstellung und Anforderungsanalyse. Das Ziel dieser Analyse ist das Reengineering vorhandener Funktionalität für unterschiedliche Vertriebswege. Zuletzt werden die funktionalen sowie die nicht funktionalen Anforderungen formuliert.

Im fünften Kapitel dieser Arbeit werden die designtechnischen Überlegungen angestellt, die zum Entwurf des neuen SOA-Modells notwendig sind (Kapitel 5). Dabei besteht das Hauptziel darin, ein System zu schaffen, das möglichst zuverlässig und einfach erweiterbar ist. Dazu wird ein Überblick über die Systemarchitektur des E-commerce-Unternehmens gegeben, in dem auch noch einmal die wesentlichen Komponenten des Systems kurz beschrieben werden.

Zuletzt wird ein Entwurf des AdressValidators vorgestellt, der für die Evaluierung des Designkonzeptes dienen soll und die Realisierung der Anforderungen und die konkrete Einführung von SOA zeigen soll. Es wird auch dargestellt, wie sich Teile der Logik des Altsystems separieren lassen. Das Prinzip der Zuständigkeit wird Kohäsion und Auslagern von Datentransformationen in Wrapperkomponenten realisieren. Die einfache Erweiterbarkeit wird durch Nutzung von Factories ermöglicht.

Im sechsten Kapitel findet eine prototypische Realisierung, das Designkonzept zu evaluieren (Kapitel 6) statt. Dazu werden die Tools und die Software vorgestellt, die für die Realisierung der Fallstudie benötigt werden. Im Anschluss findet die Implementierung der Beispielanwendung unter Verwendeung des JBoss ESB statt.

Kapitel 7 (Fazit) fasst die Resultate und Erkenntnisse zusammen, die im Rahmen dieser Arbeit erarbeitet worden sind und gibt einen Ausblick auf die Erweiterungsmöglichkeiten.

2 Technische Grundlagen

„Once you go SOA

everything becomes interoperable“

Thomas Erl[1]

Dieses Kapitel beschäftigt sich mit den technischen Grundlagen, die in dieser Arbeit benötigt werden. Dabei werden zuerst die Grundlagen von SOA zusammengefasst. Zunächst wird erörtert, was unter einer SOA zu verstehen ist und welche Rolle Services in einer SOA spielen.

Desweiteren werden die Schichten einer SOA definiert, um herauszufinden, welche SOA- Hilfskomponente, bzw. Schicht am besten für das E-Commerce Unternehmen geeignet ist. Im zweiten Abschnitt dieses Kapitels werden die Standards von Webservices erklärt, um die Vision einer SOA-Architektur zu realisieren.

Es wird gezeigt, dass die Serviceschicht die Basisarchitektur ist und als Voraussetzung für den Einsatz von SOA gilt. Dazu wird angeführt, warum die Orchestrierungsschicht für das E-Commerce-Unternehmen nicht relevant ist. Bei der Integrations-Architecture-Schicht wird der Einsatz des ESB untersucht und erklärt, inwiefern der ESB zur Lösung der Integrationsproblematik beitragen kann.

[1] Erl. (2005b)

2.1 SOA

Als Voraussetzung für die Service-Orientierung in Unternehmen steht an erster Stelle, einzelne Anwendungs-Komponenten als Dienste anzubieten, die sprach- und plattform-neutral beschrieben sind. Dies hat den Vorteil, dass Services nicht nur intern verwendet werden können, sondern auch extern, z. B von Geschäftspartnern, unabhängig davon, auf welcher Technologie bzw. Plattform die einzelnen Systeme basieren Papazoglou und Heuvel (2007a). So lassen sich durch Realisierung von Services aus Enterprise-Applikationen, die sich weder physikalisch noch technisch gleichen, neue dienstorientierte Anwendungen bauen. Demzufolge kann es durchaus Fälle geben, in denen sich einzelne Aktivitäten eines Geschäfts-Prozesses aus Services aus unterschiedlichen Betriebs-Anwendungen zusammensetzen. Es gibt keine allgemein akzeptierte Definition von SOA.Im Folgenden sind zwei mögliche Definitionen von SOA angeführt:

- Definition der SOA nach [OASIS] *A paradigm for organizing und utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measureable preconditions and expectations.*

- Definition der SOA nach HESS (2005): *„SOA ist ein Architektursmuster, das den Aufbau einer Anwendungslandschaft aus einzelnen fachlichen, das heißt geschäftsbezogenen Komponenten, beschreibt. Diese sind lose gekoppelt, indem sie einander ihre Funktionalität in Form von Services anbieten..“*

Der Begriff Anwendungs-Landschaft bezeichnet nach HESS (2005) die Gesamtheit der IT-Anwendungen eines Betriebes mit ihren Vernetzungen und Schnittstellen. In BIEN (2006) wird erklärt, dass die Idee der SOA nichts Neues, sondern eher das Bestreben einer perfekten Anwendungs-Landschaft ist, welche bei jedem Projekt angestrebt werden sollte. Dabei unterstreicht er die Fachlichkeit und nicht die Technologie. Ein weiterer Bereich, in dem die Service-Orientierung ihren Einsatz findet, ist die Modernisierung von bestehenden Systemen (Legacy-Systeme). So wird versucht, die Neuentwicklung dieser Systeme, welche mit sehr hohe Kosten verbunden ist, zu vermeiden. Die Idee hierbei ist es, Services aus dem Legacy-System zur Verfügung zu stellen, die weiterhin in die neue Umgebung integriert werden können Papazoglou und Heuvel (2007a).

Um SOA besser zu verstehen wird im folgenden Abschnitt die Kernidee von SOA vorgestellt. Dabei besteht eine SOA Architektur aus Diensten, die unterschiedliche Rollen besitzen. Diese Rollen werden im nächsten Abschnitt definiert und genauer erklärt.

2.1.1 Kernidee einer SOA

Die Kernidee einer SOA besteht drin, Services bereit zu stellen, welche von vielen Interessenten gefunden, bzw. verwendet werden können. Dadurch entstehen drei Rollen einer SOA Architektur. Abschnitt 2.1.1 beschreibt die Rollen einer SOA-Architektur.

Rollen einer SOA

Eine dienste-orientierte Architektur beinhaltet im Wesentlichen drei Rollen.

- **Dienst-Anbieter („Service Provider“):** Bereitstellung der Schnittstelle auf eine bestimmte Funktionalität.
- **Dienst-Nutzer („Service Consumer“):** Klient, der auf den Dienst zugreift.
- **Verzeichnis-Dienst („Service Broker“):** Verwaltet Informationen über Dienste und bietet Suchmöglichkeiten im Zusammenspiel von Dienst-Anbieter, Dienstnutzer und Verzeichnisdienst

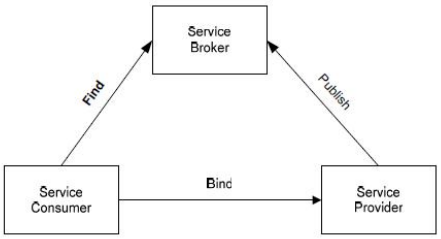

Abbildung 2.1: Die drei Rollen der Webservices

Die SOA basiert auf der Interaktion zwischen drei wichtigen Rollen, Service Provider, Service Consumer und Service Broker. Ein Dienst wird vom Dienstanbieter in einem Verzeichnisdienst veröffentlicht. Dabei werden Informationen über die Schnittstelle, den Ort und die Funktionalität des Dienstes mitgeliefert. Mit Hilfe des Verzeichnisdienstes soll dann ein konkreter Dienst entdeckt und gebunden werden, um dessen Funktionalität zu nutzen. Dabei ist die Hauptrolle der Service Provider, die Implementierung und Veröffentlichung der Services, ganz unabhängig von der Programmiersprache und der Laufzeitumgebung. Wichtig ist hierbei, dass der Dienst eine klar beschriebene Schnittstelle hat. Darüber hinaus sollte der Dienst im Netz, bzw. Netzwerk eine eigene eindeutige Adresse besitzen, um den Zugriff zu gewährleisten. Ein Service-Nutzer kann nur dann einen bestimmten Service verwenden, wenn ihm die

benötigten Informationen (Schnittstelle und Adresse) bekannt sind. Ist dies nicht der Fall, so muss es über den Service Broker geschehen. Sobald die Informationen vom Broker angekommen sind, kann der Service-Nutzer durch einen Request den Nutzungsprozess fortsetzen. Bei einem Service Broker handelt es sich um einen Vermittler zwischen dem Service-Anbieter und -Nutzer. Er hat zwei Aufgaben: zum einen Services, die von Service Provider publiziert werden, entgegenzunehmen und in einen Repository abzulegen, zum anderen über Suchfunktionen Service-Nutzern die Möglichkeit zu geben, ihre gewünschten Services zu finden. Die Nutzung eines Service Brokers ist nur dann wichtig, wenn sich die anderen zwei Rollen nicht kennen. Daher ist ein Service Broker nicht unbedingt notwendig. Die Folgende Abbildung illustriert die Kommunikation zwischen den einzelnen Webservices-Rollen. Dieses Paradigma wird als „find, bind and execute“ bezeichnet Papazoglou und Heuvel (2007a).

Das Ziel einer SOA ist es, Services bereitzustellen, welche von vielen Interessenten benutzt werden können, ganz unabhängig, welche Technologie dahinter steckt. Somit spielt die Interoperabilität bei einer SOA eine wichtige Rolle. Um diese zu gewährleisten, sind bestimmte Voraussetzungen, zu erfüllen. Diese Voraussetzungen werden im nächsten Abschnitt beschrieben.

2.1.2 SOA Schichten

In diesem Abschnitt werden die Schichten eines SOA-Modells beschrieben. Die folgende Abbildung illustriert die SOA-Schichten. Hierbei handelt sich um eine verteilte IT-Anwendungslandschaft, welche durch bestimmte Technologien und Standardisierungen ihre Dienste in die Architektur von SOA stellt. Die unterste Ebene stellt Hardware, Systeme und Netzwerk dar, auf denen sich die Anwendungen befinden. Die zweite Ebene von unten, als „Applications“ gekennzeichnet, illustrieren die Anwendungen, welche sich an unterschiedlichen Orten befinden können und auf unterschiedlichen Plattformen (J2EE/.Net) basieren können HESS (2005).

In der Service-Ebene befinden sich die Services und alles, was zu deren Verwaltung benötigt wird. Die Orchestrierungsschicht stellt das Zusammenspiel der Services dar, die oberste Ebene schließlich die Präsentationsschicht HESS (2005).

Nun werden die einzelnen Sichten der SOA beschrieben. Die Beschreibungen sind zum Teil an LIEBHART (2007) angelehnt. Der Autor beschreibt die einzelnen Schichten einer SOA und stellt einige Open-Source Produkte vor, mit deren Hilfe die einzelnen SOA-Schichten realisiert werden können.

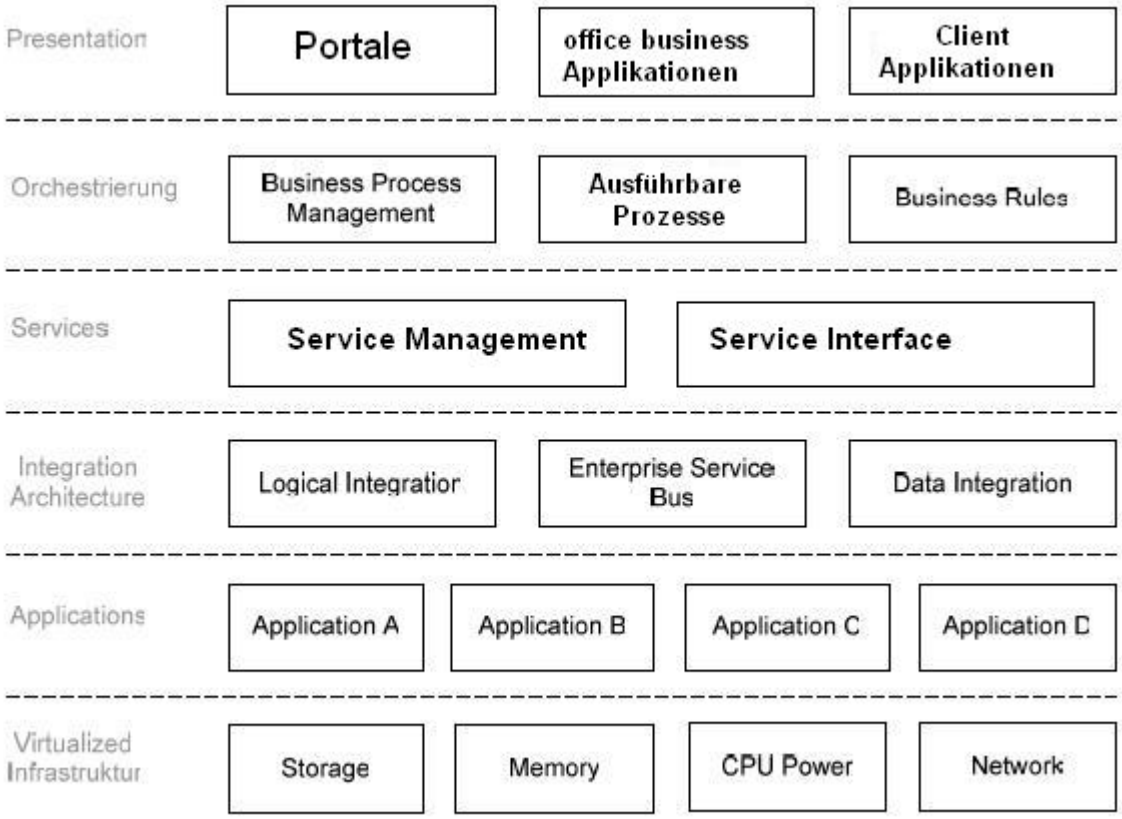

Abbildung 2.2: Die Bestandteile des SOA-Modells

Präsentationsschicht

Die Benutzerschnittstellen einer Anwendung auf der Basis von SOA unterscheiden sich nicht von einer gewöhnlichen Applikation. Dabei kann es sich um Client Applikationen (Webbasierte und Rich Client) oder Office Applikationen handeln. Jedoch sehen die SOA Blueprints der meisten Hersteller Portlets als User Interface vor LIEBHART (2007). Hierzu gibt es den Standard Web Service for Remote Portlets (WSRP). WSRP erweitert den WSDL, um den Service mit Hilfe von Portels darstellen zu können.

Orchestrierungsschicht

Als Orchestrierung wird die Kombination einzelner Services bezeichnet, die zusammen die Geschäftsprozesse darstellen. So kann beispielsweise ein orchestrierter Geschäftsprozess aus mehreren Services bestehen und selber als ein Service verwendet werden. Bei der Realisierung einer Orchestrierungsschicht spielt die Modellierung eine wichtige Rolle. Hier wird der Prozess mit Hilfe eines Ablaufdiagrames modelliert. Für die Modellierung gibt es spezielle Standards wie Business Process Execution Langauage (BPEL) und entsprechende Werkzeuge. So lassen sich beispielsweise mit Hilfe eines BPEL Tools Geschäftsprozesse modellieren und als BPEL-Datei ablegen. Diese kann dann in eine BPEL Engine geladen und ausgeführt werden LIEBHART (2007).

Auf diese Weise lassen sich aus modellierten Prozessen Workflows generieren, die dann auf jeder BPEL-fähigen Workflow-Engine ausführbar sind. In LIEBHART (2007) ist dokumentiert, dass bei einer SOA ausführbare Prozesse (Workflows) und ausführbare Geschäftsregeln neben den standardisierten Services die wichtigste Grundlage bilden.

Bei dem Einsatz von SOA für das E-Commerce-Unternehmen wird die Orchestrierungsschicht nicht eingesetzt, da das Unternehmen seine eigene Prozess Engine im Einsatz hat, wie in der Motivation beschrieben wird.

Serviceschicht

Bereits in den vorigen Abschnitten wurde erwähnt, dass die Kernidee einer SOA die Bereitstellung von Services ist und dass diese die Fachlichkeit unterstreichen und plattformneutral sein sollen. Die Idee dabei ist, Dienste einer Anwendung, bzw. von Anwendungskomponenten durch standardisierte Schnittstellenbeschreibung, wie etwa WSDL, für Webservices zu publizieren. Die eigentliche Implementierung bleibt somit nach außen verborgen und ist nur über diese Schnittstellenbeschreibung, welche sprachneutral dargestellt ist, ansprechbar. Dieses Paradigma ist in der Informatik bereits ohne SOA unter dem Begriff Information-Hiding verbreitet, wobei die sprachneutralität nicht im Vordergrund stand, sondern die saubere Art und Weise modulare Anwendungen zu entwickeln LIEBHART (2007). Der Betrieb einer SOA wird durch Service Management unterstützt.

Integration Architecture-Schicht

Bei dieser Ebene handelt es sich um eine Integrationsinfrastruktur zu Verknüpfung und Verbindung bestehender Anwendungen oder Datenbanken sowie der Koppelung von Services mit den Bestandteilen der Presentations-Ebene. Die entsprechenden Mechanismen können

entweder als logische Integration(ohne ESB), als Enterprise Service Bus oder als Datenintegration realisiert werden HESS (2005). Bei der logischen Integration wird ein Netzwerk als Infrastruktur vorausgesetzt. Die logische Integration als Realisierung einer Integrationsarchitecture setzt keine Middleware als Integrationskomponente ein. Sämtliche Komponenten der auf SOA basierenden Lösung interagieren direkt miteinander. Dadurch wissen alle Bestandteile, wo ein zu verwendender Service zu finden und wie er aufzurufen ist. Diese Variante ist für viele kleinere und mittlere auf SOA basierende Lösungen ausreichend. Dadurch werden Aufwand und Lizenskosten für den Einsatz einer gebührenpflichtigen ESB gespart.

Der Einsatz von ESB oder von einer Data-Integration-Plattform als Integration Architekture ist vor allem für größere SOA-Realisierungen sinnvoll. Dabei stellt der ESB die Kommunikation zwischen verschiedenen Services sicher, transformiert, validiert und aggregiert Meldungen und übernimmt das Routing von Meldungen David A Chappell; HUMM (2004). Dazu kann der ESB proprietäre Service- Schnittstellen als Webservice-Schnittstellen darstellen. Außerdem stellt jeder ESB weitere nützliche Instrumente zur Verfügung. Eine Data Integration Plattform ist ein spezialisierter ESB, der die unternehmensweite Harmonisierung und Konsolidierung von Daten unterstützt.

Der Einsatz eines ESB als zentrale Komponente zur Verbindung von Diensten ist für komplexere auf SOA basierende Lösungen sinnvoll. Dabei enthält der ESB eine Sammlung von Instrumenten zur sicheren und garantierten Meldungsübertragung, Routingmechanismen für die Verteilung von Meldungen, vorgefertigten Adaptoren für die Integration verschiedener Systeme, Management- und Überwachungstools. Auch stellt der ESB eine logische zentrale Anlaufstelle in einem verteilten System dar.

Durch den Einsatz von ESB wird die Migration / Updates / Ersetzen von Services erlaubt, bzw. vereinfacht. Der Einsatz von einem ESB in E-Commerce Systemen spielt eine sehr wichtige Rolle. Dabei wird dafür gesorgt, dass alle Aufträge in der richtigen Reihenfolge übertragen und abgearbeitet werden können David A Chappell; HUMM (2004). Hierfür können die einzelnen Komponenten von mehreren Applikationen verwendet werden (z. B: telefonische und online Bestellsysteme können dasselbe Addressvalidierungstool nutzen).

Applikationsschicht

Hierbei handelt sich um die Enterprise Applikationen, die ein verteiltes System abbilden. Diese Applikationen können neue oder bereits vorhandene Anwendungen, sowie Datenbanken oder anderen Datenquellen eines Unternehmens sein.

2.1.3 Zusammenfassung

Die SOA gilt seit einigen Jahren als Best Practice für diese Integration. Dabei stellt sich immer die Frage, in welcher Form der Service Provider die Nachricht (Service Request) erhalten soll - oder, genauer ausgedrückt, wie die Kommunikation zwischen den einzelnen Services ablaufen soll. Passende Koppelungstechnologien müssen entsprechend plattform- und programmiersprachenunabhängig sein und dürfen nur sehr geringe Annahmen über die integrierten Systeme machen. Von daher ist eine lose Koppelung notwendig.

Web Service Oriented Architecture ist eine mögliche Realisierung einer SOA Architecture durch den Einsatz von Webservices. Bei der Einführung der SOA-Architecture in den Beispiel-Unternehmen werden WSMO eingesetzt Papazoglou und Heuvel (2007a). Abbildung 2.3) zeigt die Web Services Standards, die für die Realisierung einer Web Service Oriented Architecture benötigt werden.

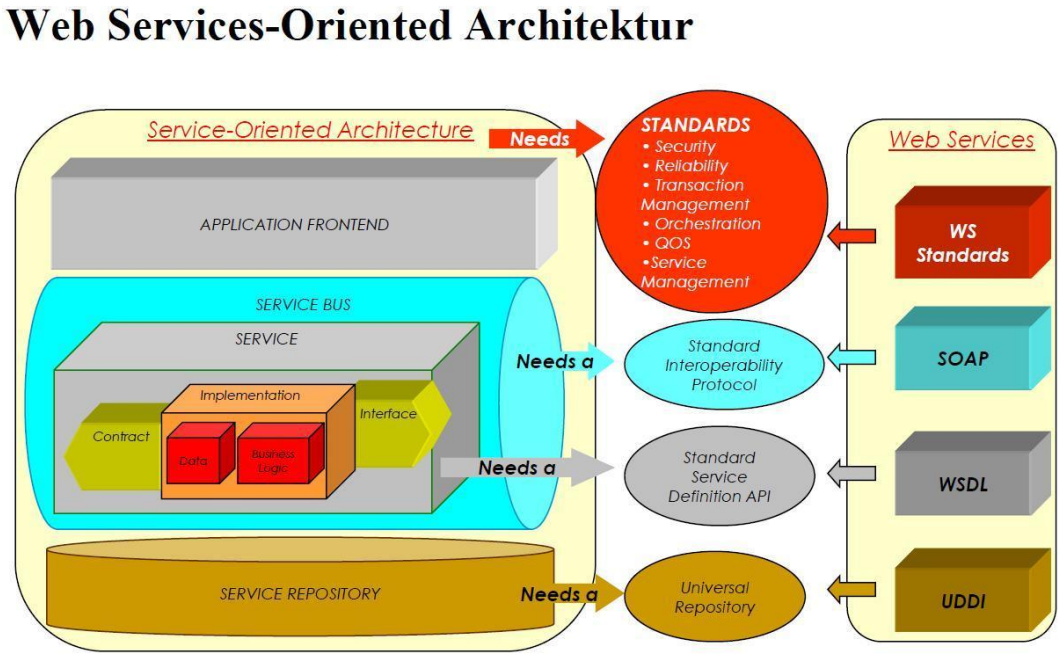

Abbildung 2.3: Web Services-Oriented Architektur

Eine Web Services-Oriented Architektur ist eine möglische SOA-Implementierung, die Web Services nutzt Papazoglou und Heuvel (2007a). Im folgenden Abschnitt findet eine Beschreibung der Web Service Technologie statt.

2.2 Webservices

Webservices sind in den letzten Jahren sehr populär geworden. Durch ein SOAP Protokoll gewährleisten Webservices die Kommunikation zwischen den Services und ermöglichen, heterogene Systeme miteinander zu kombinieren. Dabei wird auf drei standardisierte Technologien gesetzt, die in diesem Abschnitt beschrieben werden CHAPPEL (2003). Bei diesen Technologien handelt es sich um Simple Object Access Protocol (SOAP) um die Verbindung zwischen dem Dienstanbieter und dem Servicenutzer durchzuführen, Web Service Desription Language (WSDL), um Services zu beschreiben und Universal Description Discovery Integration (UDDI), um die Services zu publischen.

2.2.1 SOAP-Webservices

SOAP ist eine vom World Wide Web Consortium (W3C) standardisierte Verpackungsstruktur, um XML-basierte Dokumente über standardisierte Internet-Technologien, wie etwa SMTP, FTP und http, zu transportieren. Dadurch ist SOAP heute sehr mächtig, aber auch entsprechend komplex. Der Erfolgsfaktor von SOAP war ursprünglich die Einfachheit und Eleganz. Aber durch die neuen Standards und APIs, die hinzugekommen sind, hat SOAP seine Einfachheit verloren und seine Komplexität entsprechend erhöht. SOAP stand früher für Simple Object Access Protocol. Inzwischen wird SOAP nicht mehr als Akronym, sondern als Eigenname verwendet CHAPPEL (2003).

Die Grundidee von SOAP-Webservices ist, einen Request in Form einer XML-Datenstruktur an einen definierten Server zu schicken, wobei der Server einen Dispatcher enthält, der die XML-Datenstruktur interpretiert und die aufzurufende Operation am Server erkennt. Die benötigten Parameter für die Operation werden aus den XML-Daten ausgelesen und dem Operationsaufruf mitgegeben.

SOAP, WSDL und UDDI spielen die Zentrale Rolle für SOAP Webservices. Dabei arbeitet das Simple Object Access Protocol (SOAP) mit XML-Syntax und stellt die Verbindung zwischen Client und Webservice her. Dieses Protokoll bietet „Remote-Procedure-Calls“ (RPC) sowie den Austausch von XML-Nachrichten. Die Schnittstellen-Definitionen eines Dienstes werden mittels Web Services Description Language (WSDL) beschrieben. Universal Description, Discovery and Integration (UDDI) sind ein Industrievorschlag für verteilte web-basierende „Dienstekataloge“, die als Webservices bereitgestellt werden. Er basiert auf XML und SOAP und stellt ein Verzeichnis von Adress- und Produktdaten sowie Anwendungsschnittstellen der verschiedenen Webservices-Anbieter zur Verfügung. Ein Dienstanbieter meldet sich per UDDI bei einem Verzeichnisdienst an. Die Nutzung des Verzeichnisdienstes durch den Dienstnutzer läuft auch über UDDI CHAPPEL (2003).

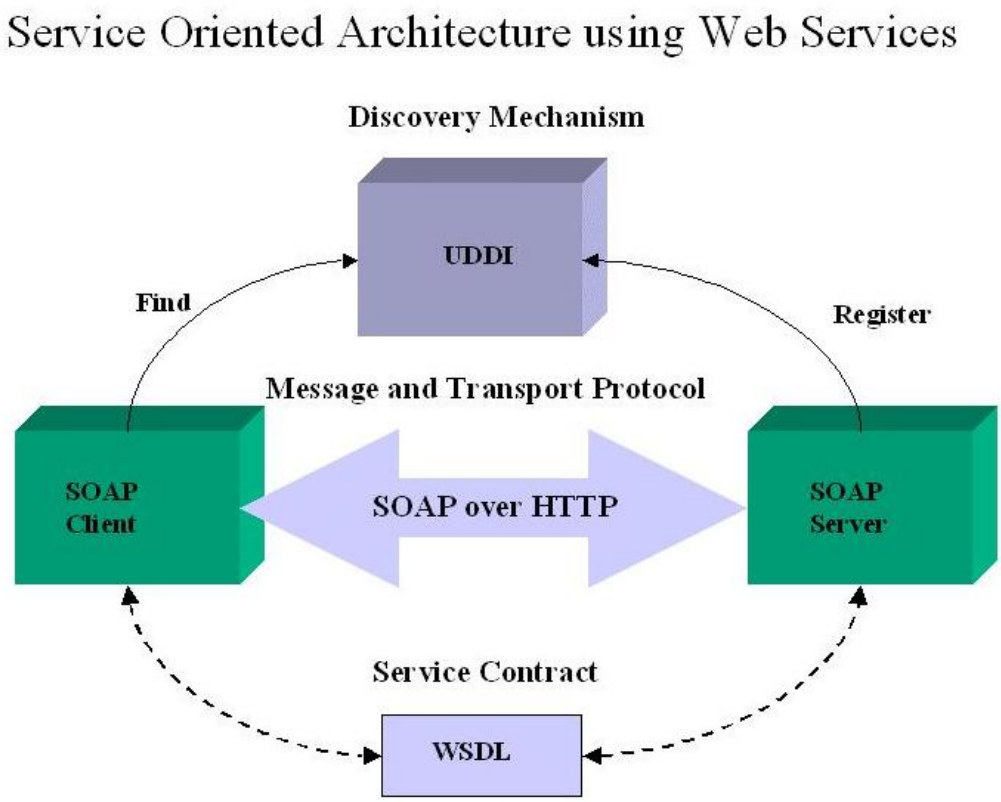

Abbildung 2.4: Webservice Architektur

Webservices werden meist mit SOAP in Verbindung gebracht. Thomas Roy Fielding, einer der Hauptautoren, die die Spezifikation Hypertext Transfer Protocol (HTTP) eingeführt haben, beschreibt in seiner Dissertation einen Architekturstil, den er REpresentational State Transfer oder kurz REST nennt. Dabei handelt es sich um eine weitere Alternative für die Realisierung von Webservices. REST baut auf Prinzipien auf, die in der größten verteilten Anwendung überhaupt eingesetzt werden - dem World Wide Web.

2.3 Zusammenfassung

SOA ist ein Entwurfsmodell, welches auf dem Konzept basiert, Anwendungslogik in Services, die über ein übliches Kommunikationsframework interagieren, zu kapseln.

In diesem Abschnitt wurde gezeigt, dass die Servicesschicht die Basis einer SOA-Architektur ist und als Voraussetzung für den Einsatz der SOA gilt. Daraus ergibt sich der Vorteil, dass dieselbe Funktionalität in verschiedenen Anwendungen idealerweise nur einmal entwickelt wird und dann als Service zur Verfügung gestellt wird, was ein wesentlicher Schritt zu schnellerer und weniger fehlerbehafteter Software-Entwicklung ist. Außerdem lässt sich durch die Kapselung von zentralen Geschäftsfunktionen in Services und die Wiederverwendung dieser Services die **Wartbarkeit** der gesamten Unternehmens-IT erhöhen. Anpassungen an die Programme, die Geschäftsfunktionen implementieren, müssen nur noch einmal zentral vorgenommen werden und nicht in vielen einzelnen Anwendungen.

Bei der Integration Architecture-Schicht wurde der Einsatz des ESB untersucht und erklärt, inwiefern der ESB zur Lösung der Integrationsproblematik beitragen kann. Dazu wurde aufgezeigt, dass die ESB-Integration weiterhin zusätzliche Vorteile für eine SOA-Architektur bringt, wobei die Komplexität verringert wird, besonderes beim Legacy-System. Alte Komponenten können übernommen und als Services verwendet werden. Grundsätzlich werden alle Komponenten integrierbar und austauschbar mit sehr geringem Aufwand, dadurch wird die Wartbarkeit des Systems vereinfacht. Durch das Angebot der der synchronen und asynchronen Kommunikation zwischen den Komponenten, sorgt der ESB dafür, ein zuverlässiges System zu haben, in dem alle Prozesse ausgeführt werden, auch wenn die Diensanbieter ausgelastet sind.

Durch den Einsatz des ESB werden die einzelnen Komponenten nur noch durch den ESB integriert und können von unterschiedlichen Anwendungen verwendet werden ohne Neuimplementierung bzw. Anpassung der Schnittstellen. Dadurch wird eine geringe Komplexität der IT-Systeme des E-Commerce-Unternehmens erreicht, besonders bei der Integration von neuen Komponenten.

Auch wurde gezeigt, warum die Orchestrierungsschicht für das E-Commerce Unternehmen nicht relevant ist.

Im nächsten Abschnitt werden die vorhandenen Ansätze zur Lösung von allgemeinen Integrations-Aufgaben aus der Literatur sowie aus der Praxis untersucht.

3 Vergleichbare Arbeiten

Im Folgenden wird aufgeführt, welche theoretischen oder praktischen Ansätze bereits zur Lösung von allgemeinen Integrations-Aufgaben und die darauf aufbauende Migrationsproblematik existieren. Es werden in zwei Abschnitten Lösungen zur Problemstellung aus der Literatur (Kapitel 3.1) sowie aus der Praxis (Kapitel 3.2) aufgezeigt. Der erste Teil dieses Kapitels beschäftigt sich mit der Vorstellung der drei bekanntesten Integrationsansätze. Dabei geht es hauptsächlich darum, zu entscheiden, welche Komponente oder Teilkomponente aus den Legacy Systemen bei einer Integration übernommen werden kann. Beim ersten Einsatz wird die existierende Applikationsinfrastrukture inklusive alle Legacy Systeme beibehalten, dabei werden nur die Schnittstellen neu implementiert. Beim zweiten Ansatz wird die alte Applikationsinfrastruktur analysiert und komplett neu entwickelt. Der dritte Ansatz ist eine Mischung der beiden Ansätze, dabei werden nach einer Analyse der existierende Applikationsinfrastrukturen nur Teilkomponenten neu entwickelt. Es wird gezeigt, dass das Re-Engineering von Funktionalitäten, die externe Dienste aufrufen, der Wrapping Ansatz, am besten geeignet ist. Für Funktionalitäten mit großem Anteil an E-Commerce Logik, wird das „Neuentwickeln von Teilkomponenten“ als bestgeeigneter Ansatz erklärt.

Nach der Vorstellung der existierenden Integrationsansätze werden im ersten Abschnitt dieses Kapitels die existierenden Migrationsansätze erklärt. Dabei werden die einzelnen Schritte definiert, die die Migration einer bestehenden Applikation in einer SOA-Architektur benötigt.

Im Allgemeinen existieren drei bekannte Migrationsprozesse, um die Verfügbarkeit des Altsystems zu gewährleisten, während das neue System mit Daten versorgt wird. Der erste Ansatz „Der Cold-Turkey-Ansatz“ versucht das Altsystem weiterhin zu nutzen, während das Neusystem entwickelt wird. Beim zweiten Ansatz, dem „Chicken-Little-Ansatz“, wird das Altsystem Stück für Stück durch das neue System abgelöst unter Verwendung von Gateways. Den dritten Ansatz, den „Butterfly-Ansatz“, kann man als reine Datenmigration betrachten. Hierbei wird das Neusystem in einer Testumgebung entwickelt, ohne dabei den alltäglichen Systembetrieb der Altanwendung zu beeinflussen oder zu stören. Im zweiten Teil dieses Kapitels werden zwei bekannte Lö-

sungen zur Problemstellung aus der Praxis vorgestellt. Es handelt sich um die SOA-Umsetzung bei der Deutschen Post und T-COM (Kapitel 3.2).

3.1 Ansätze aus der Literatur

Bei der Einführung von SOA existiert meistens eine alte Applikationsinfrastruktur, die viele Legacy Systeme beinhaltet. Diese Systeme beinhalten eine große Menge von wertvollen Funktionen und Business-Logik, die bei der Einführung von SOA berücksichtigt werden müssen.

Im ersten Abschnitt werden die Ansätze vorgestellt, die eine Möglichkeit anbieten, die neue SOA-Architektur einzuführen, ohne die wertvollen Informationen zu verlieren. Bei der Integration neuer Applikationen und beim Re-Engineering alter Applikationen sind meistens Migrationsprozesse notwendig, um die Neusysteme mit Daten zu versorgen. Der zweite Abschnitt dieses Kapitels beschäftigt sich mit der Untersuchung der möglichen Migrationprozesse, die hauptsächlich für die Verfügbarkeit des Altsystems sorgen, während das Neusystem mit Daten versorgt wird. Es wird gezeigt, dass das der Cold-Turkey-Ansatz am besten für E-Commerce Systeme geeignet ist.

3.1.1 Integrationsansätze

Bestehende Anwendungen werden als wertvolle Vermögenswerte gesehen und sollten nicht einfach weggeworfen werden. Sie beinhalten wichtige Funktionalität, die als Services bei der Einführung einer SOA gemappt werden können. In der Literatur über allgemeine Integrationsaufgaben existieren hauptsächlich drei unterschiedliche Integrationsansätze. Im ersten Ansatz wird die existierende Applikationsinfrastruktur beibehalten und nur die Schnittstellen als Webservices entwickelt (Kapitel 3.1.1). Innerhalb des zweiten Ansatzes werden alle Legacy-Systeme neu entwickelt und in Services integriert (Kapitel 3.1.1). Der dritte Ansatz ist eine Mischung aus den zwei ersten Ansätzen, bei der nur Teilkomponenten des Legacy-Systems neu entwickelt werden (Kapitel 3.1.1).

Aufbewahrung bestehender Applikations-Infrastruktur (Wrapping)

Bestehende Anwendungen enthalten eine große Menge von wertvollen Funktionen und Business-Logik, die bei der Einführung von SOA berücksichtigt werden müssen. Das Mapping existierender Applikationen in Services ist der schnellste und einfachste Ansatz für die Einführung von SOA. Bei diesem Ansatz werden nur die Schnittstellen von Legacy Systemen untersucht und durch Services-Schnittstellen ersetzt, meistens durch Webservices Schnittstellen, die mittels WSDL beschrieben werden Bisbal u. a. (1997). Der Source Code wird nicht

einmal betrachtet. Die Schnittstellen der Legacy Systeme sind meistens Datenschnittstellen der alten Programme. Die Integration findet auf der binären Ebenen statt. Die Webservices werden als Proxys entwickelt. Der Vorteil dieses Ansatzes ist, dass die alten Programme in ihrer ursprünglichen Umgebung bleiben dürfen. Man spart die Kosten und die Risiken einer Migration.

Der Nachteil ist, dass Web Services, die aus diesem Ansatz entstehen, ineffizient und unflexibel sind, weil sie eine Black Box [1] Schnittstelle anbieten. Der Anwender wird gezwungen, immer mehr Codes um sie herumzubauen. Die Lücke zwischen den Anforderungen der Web Services und der Leistung des alten Codes wird immer breiter, ohne die Möglichkeit, den alten Code verändern zu können. [Bisbal u. a. (1999)]

In der Literatur wird dieser Ansatz oft als schnellster und beliebter Ansatz mit einem geringen Aufwand angesehen. Aber es wird häufig darauf hingewiesen, dass dieser Ansatz möglicherweise negative Auswirkungen auf zukünftige System-Wartungen und Erweiterungen, haben kann, weil die Struktur der originalen Systeme nicht untersucht wird, sondern nur die Schnittstelle und die Legacy Systeme bleiben als Black Box.

Bisbal erklärt, dass Legacy-Informationssysteme meistens das Basisnetz des Unternehmens sind und kritisch für die Geschäftsprozessausführung. Sie werden angesetzt, um die negativen Auswirkungen auf zukünftige System-Wartungen und Erweiterungen zu vermeiden.Wu u. a. (1999a)

Zhuopeng Zhang (2004) erklärte, dass Legacy-Systeme ein wertvolles Kapital für die Organisationen sind. Mit Hilfe von Web Services und SOA wird es meistens erleichtert, Applikationen aus Legacy-Systemen als Services anzubieten und somit einen Übergang zu einer Service orientierten Architektur zu beschaffen. Die Legacy-Systeme sollen über einen Adapter und durch die Analyse der Legacy-Schnittstellen in Services integriert werden Zhuopeng Zhang (2004).

Lewis gibt weiter an, dass das Wrapping von Legacy Komponenten zu Services die einfachste Disziplin für die Definition der Webservices-Schnittstellen ist und bezeichnet diesen Ansatz als attraktivsten Ansatz, um Geschäftslogik in die SOA zu integrieren. Außerdem erwähnt Lewis, dass die unterschiedlichen Eigenschaften von Legacy-Systemen den Integrationsprozess komplizierter machen können und eine Analyse der Realisierbarkeit im Voraus gemacht werden soll, um unnötigen Aufwand zu vermeiden.

Canfora Gerardo Canfora (2006), hat die Black Box Integrations-Technik vorgeschlagen. Diese Technik benötigt keine Reverse Engineering [2] oder Code-Änderung, sondern beschäftigt sich

[1] Als Black Box bezeichnet man ein (möglicherweise sehr komplexes) System, von welchem im gegebenen Zusammenhang nur das äußere Verhalten betrachtet werden soll

[2] bezeichnet den Vorgang, aus einem bestehenden, fertigen System oder einem meistens industriell gefertigten Produkt durch Untersuchung der Strukturen, Zustände und Verhaltensweisen, die Konstruktionselemente zu extrahieren. Aus dem fertigen Objekt wird somit wieder ein Plan gemacht.

mit dem Wrapping des originalen Systems durch die Einführung einer modernisierten Schnittstelle, meistens einer Web Service-Schnittstelle Gerardo Canfora (2006).

Integration von Legacy-Systemen in Services (Neuentwicklung von Legacy-Systemen)

Bei diesem Ansatz geht es hauptsächlich um das Neuentwickeln von Legacy-Systemen. Dazu dient die Unterstützung der automatischen Code-Generierung für Web Services unter Betrachtung der Integration von Legacy-Systemen in Services [Zhuopeng Zhang (2004)]. Dieser Ansatz ist sehr aufwändig bei der Einführung von SOA, weil nichts aus der alten Applikationsinfrastrukture übernommen wird.

Die Neuentwicklung von Legacy-Systemen ist meistens der revolutionäre Ansatz. Wie der Name schon sagt, werden bei diesem Ansatz die Legacy Systeme komplett neu entwickelt. Dabei werden zuerst die Systeme untersucht, um die wichtigsten Geschäftsprozesse zu identifizieren. Bei der Untersuchung der Legacy Systeme gehört auch dazu, den Source Code zu analysieren und service-orientiert neu zu gestalten. Danach werden die Services erkannt, die nach außen für andere Systeme zur Verfügung gestellt werden sollen und als Webservices implementiert und mittels WSDL beschrieben. Der Vorteil von diesem Ansatz ist, dass veraltete und nicht verwendbare Methoden herausgenommen werden können Wu u. a. (1999b). Ein weiterer Vorteil dieses Ansatzes bestehen darin, dass die Entwickler nicht mehr alte oder unterschiedliche Technologien und Methoden verwenden müssen. Zhang und Yang erklärten, dass es diese Methode erlaubt, dass der Entwickler bei der Schnittstellen-Implementierung die neuesten Web Service Technologien und Programmiersprachen verwendet, Wartungskosten spart und die Effizienz der Transaktionen verbessert. Aber es ist ein revolutionärer Ansatz, der negative Auswirkungen haben kann, indem die alte Geschäftslogik nicht hundertprozentig übernommen wird Zhuopeng Zhang (2004).

Quocirca erklärt, dass ein vollständiger Ersatz der bestehenden Infrastruktur bei einer SOA Integration in der Regel nicht realisierbar ist und dass die Applikationen aus dem Legacy-Systemen in der neuen Architektur aufbewahrt werden müssen, damit sie die wichtigen Geschäftsprozesse des Unternehmens weiterhin unterstützen können.

Quocirca erklärt weiter, dass es für die Integration von SOA sehr wichtig ist, bestehende Applikationen zu untersuchen, um herauszufinden, welche Funktionalitäten doppelt implementiert sind, bzw. überflüssig werden Ltd (2007).

Für die Untersuchung und Analyse der Altsysteme wurden zwei Ansätze erwähnt, das Reverse Engineering und der hierarchische Clustering Algorithmus. Das Reverse Engineering ist ein Prozess, der das betrachtete System analysiert, um seine Komponenten und ihre Beziehungen zu identifizieren. Zweck dieses Prozesses ist es, für das analysierte System neue Abbildungen des Systems zu schaffen, meist in anderer Form und auf höherer Abstraktionsebene.

Reverse Engineering ist hierbei der erste Arbeitsschritt im Re-Engineering-Prozess. Der hierarchische Clustering Algorithmus gruppiert die Objekte einer Datenbank in bekannte Gruppen, auch Cluster genannt, sodass zwei Objekte aus einem gleichen Cluster möglichst ähnlich zueinander und zwei Objekte aus unterschiedlichen Clustern möglichst unähnlich zueinander eingeteilt werden. Das Reverse Engineering sieht vor, dass der Algorithmus vom Code abgetrennt wird. Durch eine Analyse des Codes wird die darin enthaltene Logik modelliert und in Form von Diagrammen, z. B. Aktivitätsdiagrammen, Sequenzdiagrammen oder Zustandsdiagrammen beschrieben. Auf einer höheren Abstraktionsebene wird beschrieben, was in dem Code geschieht und die Diagramme können als Grundlage für die Reimplementierung des Codes in einer anderen Sprache dienen.

Zhuopeng Zhang (2004) hat einen Ansatz vorgeschlagen, der einen hierarchischen Clustering Algorithmus [3] verwendet, um mögliche Servicekonditaten zu identifizieren Zhuopeng Zhang (2004).

Der Vorteil dieses Ansatzes ist die Wiederverwendung der alten Lösung ohne den alten Code zu nutzen. Der Nachteil ist die Notwendigkeit, die alte Lösung neu zu kodieren und zu testen. Es kann lange dauern, bis der neue Code die gleiche Reife erlangt wie der alte.

Neuentwicklung von Teilkomponenten des Legacy-Systems

Bei diesem Ansatz handelt es sich um eine Mischung der beiden vorgestellten Ansätze, wobei die Legacy Systeme nur teilweise neu entwickelt werden, indem die Legacy Systeme und der Source Code analysiert und serviceorientiert gestaltet werden. Funktionalitäten, die zweimal implementiert werden, bzw. überflüssig sind, werden entfernt oder zusammengeführt. Hierbei wird mit dem alten Sourcecode weitergearbeitet. Er wird in eine Form versetzt, die es erlaubt, den Code in die neue Umgebung zu übertragen und ihn dort neu zu kompilieren und als Web Services einzubinden. Vorher muss der Source Code allerdings transformiert werden, damit er wiederverwendbar wird. Das kann auch Kosten verursachen, vor allem wenn dies manuell erfolgen muss. Der große Vorteil dieses Ansatzes ist, dass die Integration auf der Sourceebene vollzogen wird. Damit hat der Anwender mehr Möglichkeiten, den Code zu beeinflussen. Der Hauptnachteil ist die Notwendigkeit, sich mit dem alten Code zu befassen, und das ist manchmal sehr kritisch.

Zhang und Yang erklärte, dass die Neuentwicklung von Teilkomponenten ein praktischer Ansatz ist, weil der alte Source Code, bzw. die alte Geschäftslogik, aufbewahrt werden und die Entwickler bekommen einen Überblick auf den Systemen. Dadurch werden zukünftige Wartungen bzw. Erweiterungen der neuen Systeme erleichtert Zhuopeng Zhang (2004).

[3] Beim Clustering werden die Objekte einer Datenbank in (a priori unbekannte) Gruppen, auch Cluster genannt, eingeteilt werden, sodass zwei Objekte aus einem gleichen Cluster möglichst ähnlich zueinander, und zwei Objekte aus unterschiedlichen Clustern möglichst unähnlich zueinander sind

Zhan and Yang hat einen Ansatz vorgestellt, der auf die Aufdeckung wertvoller Geschäftslogik des Legacy-Codes basiert. Das geschieht durch das Reverse Engineering und Teil-Wrapping des Legacy-System-Codes. Ziel dabei ist, wertvolle und agile Services aus der bestehende Geschäftslogik schnell und zuverlässig zu schaffen Zhuopeng Zhang (2004).

Zhan and Yang argumentiert, dass dieser Ansatz die Lücke, Effektive Integration von Legacy-Systemen in SOA und Anwendungen, schließen kann, indem Teile der Legacy-Systeme unverändert und verteilt bleiben. Dabei bleiben die Entwicklungskosten niedrig und die wichtige Geschäftslogik vom Legacy System wird beibehalten Zhuopeng Zhang (2004).

Die Möglichkeit, nur Teile aus den Legacy-Systemen neu zu entwickeln, ist aber meistens die plausible Lösung. Dabei werden die Vorteile aus den zwei ersten Ansätze übernommen. Beim Wrapping-Ansatz werden nur die Schnittstellen neu entwickelt. Dadurch fordert dieser Ansatz keine genaue Analyse des Gesamtsystems und die gesamten Funktionalitäten werden beibehalten. Der Nachteil dabei ist die schwere Wartbarkeit des Systems, denn die einzelnen Funktionen werden als Blackboxes dargestellt. Bei der Neuentwicklung aller Komponenten oder der Teilkomponenten des Legacy-Systems stellt sich meistens die Frage, ob eine Migration der Daten auf der Datenbankebene stattfinden muss, denn die Datenübernahme nimmt eine gewisse Zeit im Anspruch und die Verfügbarkeit des Altsystems wird beeinträchtigt. Im nächsten Abschnitt werden die möglichen Migrationsprozesse vorgestellt, die während der Datenübernahme im Neusystem für die Verfügbarkeit des Altsystems sorgen.

3.1.2 Migrationsprozesse

In der Literatur wird häufig davon ausgegangen, dass bei der Integration einer Anwendung zu beachten ist, dass nicht notwendigerweise relationale Datenbanken genutzt werden, sondern z. B. Dateien, die im Dateisystem liegen, besonders wenn bei der Integration Änderungen auf der Persistenzschicht durchgeführt wurden. Daher ist es häufig nicht möglich, mit mehreren Programmen gleichzeitig auf dieselben Daten zuzugreifen.

Eines der zentralen Probleme, das die im Folgenden dargestellten Integrationsprozesse lösen müssen, ist daher, die Verfügbarkeit des Altsystems zu garantieren, während das Neusystem mit den nötigen Daten versorgt wird [MIG08]. Es existieren in der Literatur drei mögliche Prozesse, um das Problem umzugehen. Im Folgenden werden diese Prozesse vorgestellt:

Der Cold-Turkey-Ansatz

Ein bekannter Ansatz für das Re-Engineering von Legacy-Systemen ist der Cold-Turkey-Ansatz. Bei ihm wird versucht, das Neusystem mit modernen Mitteln unter Nutzung eines Wasserfall-Modells herzustellen. Während das Altsystem weiterhin verwendet wird, wird das Neusystem komplett entwickelt. Dabei wird erst das Altsystem vollständig analysiert, das Neusystem vollständig hergestellt und dann werden die Daten übertragen. Daraufhin übernimmt das Neusystem sämtliche Aufgaben des Altsystems, das dann nicht mehr benötigt wird.

Laut [MIG08] könnten bei dem Ansatz folgende Problemen entstehen:

- Fehlende Spezifikationen

Die einzige Dokumentation für ein Altsystem ist üblicherweise der Quellcode. Die Entwickler des Altsystems sind meistens nicht mehr verfügbar und die Dokumentation ist nicht vorhanden, schlecht geschrieben oder veraltet. Auch der im Altsystem verwandte Programmierstil entspricht meistens nicht dem aktuellen Stand der Technik. Die Gewinnung einer Spezifikation durch das Analysieren des Codes ist in diesem Fall komplex und teuer.

- Undokumentierte Abhängigkeiten

Im Laufe der Zeit können Fremdsysteme an das Altsystem angeschlossen worden sein, die auf das Altsystem zugreifen. Diese Abhängigkeiten sind im Altsystem nicht dokumentiert, müssen aber trotzdem entdeckt und beachtet werden, da sonst die abhängigen Systeme ausfallen würden.

- Zu große Altsysteme

Altsysteme müssen oft eine „100 prozentige“ Verfügbarkeit aufweisen. Allerdings kann das Übertragen der Daten aus dem Altsystem bei einem großen Datenbestand mehrere Tage oder Wochen dauern[HIL07]. Außerdem müssen die Daten für gewöhnlich noch für das neue System aufbereitet werden. Wenn in der Zeit der Datenübertragung nicht auf das Altsystem verzichtet werden kann, dann kann der Cold-Turkey-Ansatz nicht genutzt werden[HIL07].

- Die Schwierigkeit große Projekte zu managen

Typischerweise sind Altsysteme meistens sehr große Systeme. Die Entwicklung des neuen Systems wird daher nur mit einer großen Anzahl Entwickler in vertretbarer Zeit gelingen. Große Projekte sind allerdings schwierig zu handhaben und scheitern daher häufig.

- Analyse

Eine Integration nach dem Cold-Turkey-Ansatz kann erst dann beginnen, wenn eine komplette Analyse des Altsystems durchgeführt ist. Da das Altsystem aber komplex und kompliziert ist, wird immer weiter analysiert, da nicht sichergestellt werden kann, dass das gesamte System verstanden wurde. Wenn die Analyse nie abgeschlossen werden kann, kann natürlich die Integration nicht beginnen. Insgesamt wird der Cold-Turkey-Ansatz als sehr starr in Bezug auf die Anforderungen an das neue System und als mit einem hohen Risiko behaftet angesehen [HIL07]. Als Alternative wurde der im nächsten Abschnitt vorgestellte Chicken-Little-Ansatz entwickelt [STEIN08].

Der Chicken-Little-Ansatz

Der Chicken-Little-Ansatz ist ein iterativer Ansatz, der die oben aufgezählten Risiken des Cold-Turkey-Ansatzes vermeiden soll. Das Altsystem bleibt dabei im Einsatz und wird Stück für Stück durch das neue System abgelöst durch Verwendung von Gateways. Durch diesen iterativen Ansatz, sollen die oben geschilderten Probleme in kleine, überschaubare Teilprobleme zerlegt werden, die dann einzeln gelöst werden können [HIL07]. Teile der Daten des Altsystems können in das neue System übernommen werden. Dies kann, unter Einsatz eines entsprechenden Gateways, geschehen. In [BEC04] wird argumentiert, dass wenn ein Schritt der Integration fehlschlägt nur dieser Schritt wiederholt werden muss, und nicht das gesamte Projekt fehlschlägt. Die Probleme verschwinden durch diesen Ansatz nicht, aber die durch sie entstehenden Risiken für das Integrationsprojekt werden überschaubarer.

- **Gateways**

Das iterative Vorgehen soll durch Gateways ermöglicht werden. Dabei handelt es sich um Software, die die Anfragen der Benutzer auf das Altsystem und das Zielsystem aufteilt [HIL07]. Um Gateways nutzen zu können, teilt der Chicken-Little-Ansatz das Altsystem in folgende drei Schichten ein (vgl. Abbildung 3.1: Wu u. a. (1999b)).

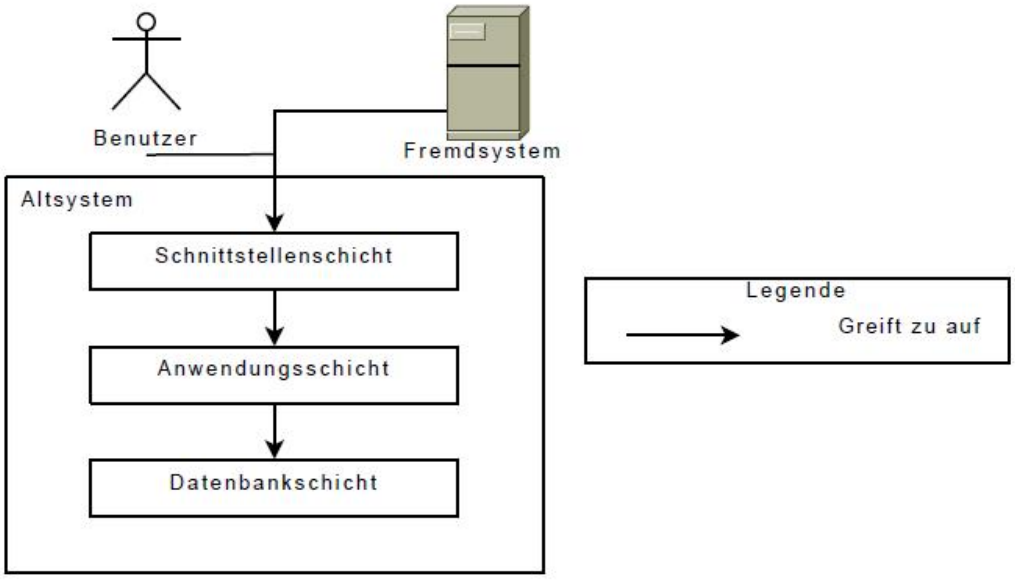

Abbildung 3.1: Schichten des Altsystems

- Schnittstellenschicht
- Anwendungsschicht
- Datenbankschicht

Gateways sind Programme, die Aufrufe auf das Altsystem und das Zielsystem aufteilen. Dabei können die Gateways zwischen den Benutzern bzw. Fremdsystemen und der Schnittstellenschicht sein oder zwischen zwei Schichten des Altsystems. In Abbildung 3.2 ist beispielhaft ein Datenbank-Gateway dargestellt. Wo genau Gateways eingesetzt werden, hängt vor allem davon ab, wie gut das Altsystem modularisierbar ist [MIG08].

- **Schritte des Chicken-Little-Ansatzes**

Der Chicken-Little-Ansatz beinhaltet 11 Schritte, in denen die Integration durchgeführt wird. Diese Schritte hängen nur teilweise voneinander ab und können auch parallel durchgeführt werden. Die Abhängigkeit der Schritte ist in Abbildung 3.3 nachvollziehbar. Diese Schritte werden in jedem Inkrement ausgeführt. Es können auch Schritte ausgelassen werden [BRODIE].

1. Das Altsystem inkrementell analysieren

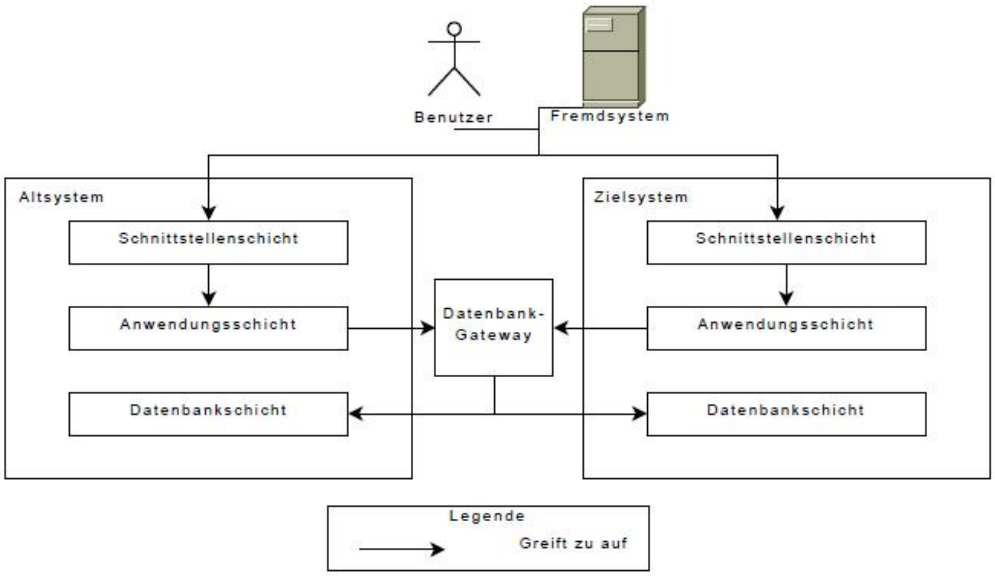

Abbildung 3.2: Beispiel eines Gateways

Es muss, wie beim Cold-Turkey-Ansatz, das Altsystem analysiert werden, um die Spezifikation des neuen Systems zu erhalten.

2. Das Altsystem inkrementell zerlegen Das Altsystem soll in einzelne Module zerlegt werden. Dazu muss es eventuell refaktorisiert werden. Das Ziel ist eine schwache Kopplung der Module, so dass sie einzeln durch das neue System ersetzt werden können.

3. Inkrementell die Zielschnittstellen entwerfen Die Schnittstellen werden entworfen und es wird entschieden, ob ein Gateway für die Schnittstellenschicht benötigt wird.

4. Inkrementell das Zielsystem entwerfen Es kann das alte System nachgebaut werden oder ähnliche Geschäftsprozesse im Zielsystem entworfen werden.

5. Inkrementell die Zieldatenbank entwerfen Das Schema für die Datenbank des Zielsystems entwerfen.

6. Die Zielumgebung inkrementell installieren Die Hardware und die Software, die für den Betrieb des Neusystems benötigt werden, installieren.

7. Benötigte Gateways inkrementell erzeugen und installieren Im Verlauf der Integrationen werden möglicherweise verschiedene Gateways benötigt. Diese können gekauft oder selbst hergestellt werden.

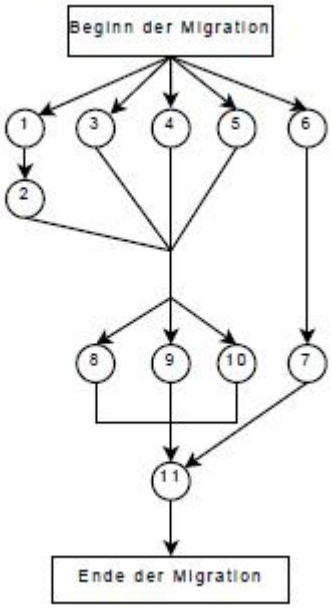

Abbildung 3.3: Schritte des Chicken-Little-Ansatzes

8. Die Datenbank des Altsystems inkrementell integrieren Teile der Daten des Altsystems können in das neue System übernommen werden. Dies kann, unter Einsatz eines entsprechenden Gateways, inkrementell geschehen. Dabei müssen die Daten aus der alten Datenbank geladen, gesäubert, konvertiert und in der neuen Datenbank gespeichert werden.

9. Das Altsystem inkrementell integrieren Entsprechend den Anforderungen an die Anwendung, werden einzelne Module integriert. Kriterien für die Reihenfolge der Integration können sein:

   - Einfache Integrierbarkeit
   - Verursachte Kosten
   - Bedeutung des Moduls für die Anwender

10. Die Benutzungsschnittstellen des Altsystems inkrementell integrieren Die Benutzungsschnittstelle wird auf die neue Version umgestellt. Um zu verhindern, dass die Benutzer zwischen der neuen und der alten Benutzungsschnittstelle wechseln müssen, kann ein Gateway eingesetzt werden.

11. Inkrementell auf das Zielsystem umsteigen

Das Zielsystem kann benutzt werden. Dabei können erst einzelne Benutzer umsteigen, um das System zu testen. Nach und nach können alle Benutzer auf das Zielsystem umsteigen.

Der Butterfly-Ansatz

In [STEIN08] wird die Benutzung von Gateways, um eine Integration inkrementell durchführen zu können, kritisiert. So ist ein inkrementelles Umsteigen auf das Zielsystem unter Risikogesichtspunkten zwar zu befürworten, allerdings erhöht die Benutzung von Gateways die Komplexität und damit das Risiko des Integrationssprojekts zu scheitern. Weitere Kritikpunkte an Gateways umfassen, dass Gateways:

- keine Unterstützung für Transaktionsmanagement bieten. Daher kann die Konsistenz der Daten zwischen dem Altsystem und dem Zielsystem nicht sichergestellt werden.
- keine Möglichkeit bieten, zwischen Unterschieden in Struktur und Repräsentation der beiden Datebankschemata zu vermitteln
- schwer zu bauen und zu betreiben sind.

Bei dem Butterfly-Ansatz wird angenommen, dass die Zusammenarbeit zwischen Alt- und Neusystem nicht zwingend notwendig ist und dass es sich um eine reine Datenmigration handelt Wu u. a. (1997). Während das Altsystem im Betrieb bleibt, findet zunächst die Datenmigration auf der Datenbankebene statt. In einer Testumgebung wird anschließend das neue System entwickelt, ohne dabei den alltäglichen Systembetrieb der Altanwendung zu beeinflussen oder zu stören. Die Butterfly-Methode benötigt im Gegensatz zum Chicken-Little-Ansatz keine Einführung von Gateways zwischen den Applikationsebenen [STEIN08].

Ein Datenbank-Gateway muss für jeden Datenbankzugriff entscheiden, ob der Zugriff auf die Datenbank des Altsystems, des Zielsystems oder auf beide erfolgen muss. Ist Letzteres der Fall, muss ein Schreibzugriff durch ein 2-Phasen-Commit-Protokoll durchgeführt werden. Die Konsistenz der Daten beim Schreiben in heterogenen Datenbanken zu garantieren ist ein komplexes Problem und daher nur mit viel Aufwand lösbar [BRODIE]. Der in [STEIN08] dargestellte Butterfly-Ansatz geht davon aus, dass Interaktion zwischen dem Altsystem und dem Zielsystem während der Integration unnötig ist Mahmoud und Gomez (2008). Daher besteht keine Notwendigkeit für die Komplexität erhöhende Gateways [STEIN08].

Mit Hilfe einer Datenzugriffskomponente und einer Transformationskomponente werden die Daten nach und nach übertragen. Während der Übertragung werden zwischenzeitliche Änderungen des Datenbestands in Temporärspeichern abgelegt, die ebenfalls in das Neusystem

übernommen werden und erst mit dem finalen Abschalten des Altsystems enden. Der Zusammenhang ist in Abbildung 3.4 dargestellt.

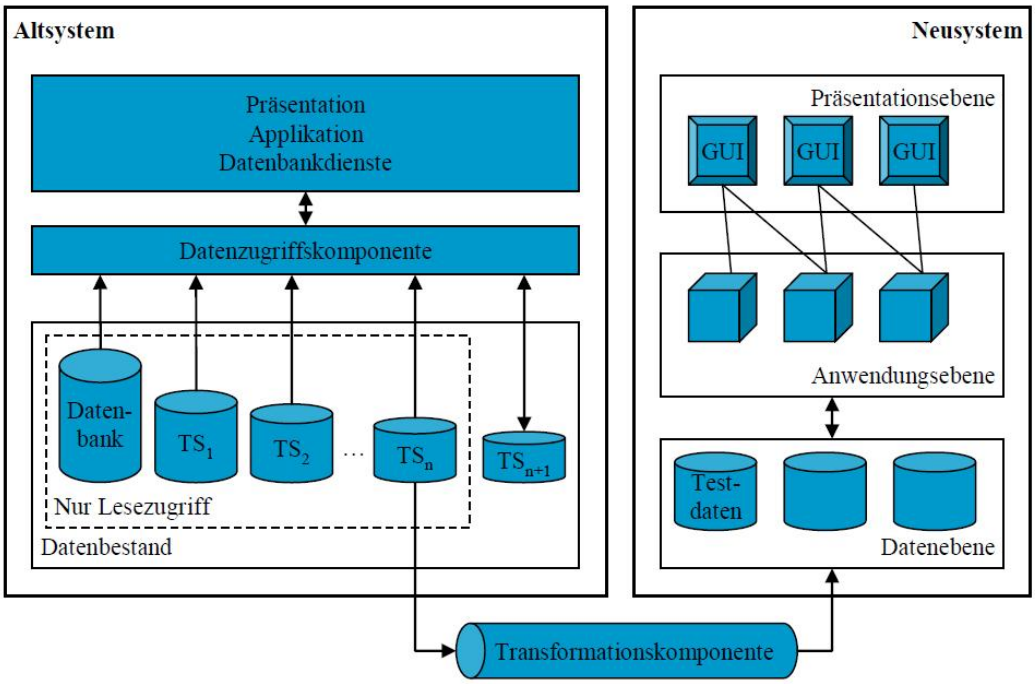

Abbildung 3.4: Butterfly-Ansatz: In Anlehnung an Wu et al. (1999), S. 3.

Das Vorgehen lässt sich dabei in sechs Phasen unterteilen, wobei der Kernpunkt der Butterfly-Methode in der Phase 4 zu finden ist. Jede Phase enthält wiederum eine Menge von Aktivitäten, die unterschiedlichen Einfluss auf den Erfolg der Integration haben. Die Aktivitäten werden dabei teilweise parallel und wiederholt durchgeführt. Teilweise ist es vorausgesetzt, bestimmte Aktivitäten abzuschließen, um mit der nächsten Aktivität zu beginnen:

1. **Phase 1**: Vorbereitung

   Sobald eine Entscheidung für eine Migration gefallen ist, können die Vorbereitungen für das Projekt anfangen. Offene und ungeklärte Fragen sollen sofort geklärt werden. Dabei werden Benutzeranforderungen festgelegt und Erfolgsfaktoren definiert. Neben der Bereitstellung der notwendigen Hardware steht die *Konzeption der Systemarchitektur* im Mittelpunkt.

2. **Phase 2**: Analyse des Altsystems und Datenmodell-Entwicklung

Für die Integration ist es sehr wichtig, das Altsystem verstanden zu haben. Dazu müssen Schnittstellen, Applikation und Datenhaltung erkannt und analysiert werden. Redundanzen sollen identifiziert werden, um die Prozesse und das System zu verschlanken. Im nächsten Schritt werden Schnittstellen festgelegt und die Funktionalität des neuen Systems beschrieben. Des Weiteren soll der Umfang der zu migrierenden Daten bestimmt werden Wu u. a. (1997). Falls Wechselwirkungen mit anderen Systemen bestehen, sollen diese Wechselwirkungen identifiziert und dokumentiert werden, um sie zu prüfen und in der Entwicklung des neuen Systems zu berücksichtigen. Neben dem Datenschema wird eine Datenzugriffskomponente entwickelt, die während des Migrationsprozesses für den Datenzugriff im Altsystem verantwortlich ist und eine Koordinations- und Verwaltungsfunktion für aktuellste Datenbestände erfüllen muss. Zum Abschluss der Phase werden Regeln für das Abbilden (Mapping) und Übertragen der Daten, festgelegt [STEIN08].

3. **Phase 3**: Testdaten bereitstellen Um die Entwicklung zu beschleunigen, werden keine Testdaten generiert, sondern direkt aus der Datenbank des Altsystems ausgewählt. So werden in der Entwicklung anhand reeller Daten Fehler schneller identifiziert und bereits in einer sehr frühen Projektphase behoben. Die Transformationskomponente wird anhand der Vorgaben der vorherigen Projektphasen entwickelt und mit Hilfe der Beispieldaten getestet. Diese Daten werden in einer Testdatenbank im Neusystem abgelegt und stehen nicht nur den Entwicklern zur Verfügung, sondern gestatten zu einem späteren Zeitpunkt Schulungen anhand realer aber nicht operativer Daten durchzuführen. Dieses Vorgehen vermeidet zum einen Fehler in der Entwicklung und führt zum anderen dazu, dass Fallstudien in Schulungen direkt auf existenten Datensätzen aus dem täglichen Arbeitsablauf aufbauen können, was wiederum zu einer verbesserten Anwendung der Erkenntnisse aus der Schulung in der Praxis führt [STEIN08].

4. **Phase 4**: Schrittweise Integration der Systemkomponenten (bis auf Daten) Die Applikationsebene des Neusystems wird gleichzeitig Neben den Benutzer- und Systemschnittstellen schrittweise entwickelt und anhand der Beispieldaten aus Phase 3 wiederholt getestet. Dabei werden mögliche wiederverwendbare Komponenten des Altsystems genutzt, um Aufwand und kosten für das Projekt zu ersparen und Fehler zu vermeiden. Nach erfolgreicher Entwicklung der Komponenten werden diese in einem nächsten Schritt in die Systemarchitektur des Neusystems integriert. Anschließend werden die Komponenten nochmals getestet und gegen die Anforderungen aus den ersten Projektphasen geprüft Wu u. a. (1997).

5. **Phase 5**: Schrittweise Datenmigration und Schulung der Anwender

   Die Datenzugriffskomponente aus Phase 2 wird in das Altsystem eingeführt und steuert ab der Einführungszeitpunkt mehrere Lese- und Schreibzugriff auf den Datenbestand.

Neben der vorhandenen Datenbank werden mehrere TempStores in der Datenebene des Altsystems eingerichtet. Die bestehende Datenbasis wird mit einem Schreibschutz versehen. Änderungen am Datenbestand werden fortan im TempStore TS1 abgelegt. Unveränderte Daten werden weiterhin von der Ursprungsdatenbank abgerufen. Der Zugriff auf aktualisierte Daten ist über den TempStore TS1 sichergestellt. Anschließend wird der Datenbestand des Altsystems über die Transformationskomponente in das neue System übertragen. Dabei wird das Datenformat mit Hilfe von Abbildungstabellen in das neue Datenmodell überführt. Die Abbildungstabellen enthalten Regeln für die eindeutige Zuordnung von Daten basierend auf den Datenmodellen der beiden Systeme. Ist die Migration der Datenbasis abgeschlossen, wird der TempStore TS1 mit einem Schreibschutz versehen. Fortan werden Änderungen des aktuellen Datenbestands im TempStore TS2 abgelegt. Nun beginnt die Überführung der Daten aus TS1 mit Hilfe der Transformationskomponente in das neue System. Wenn der Vorgang beendet ist, wird TS2 schreibgeschützt und migriert. Aktualisierungen des Datenbestandes werden dann in TS3 gespeichert. Die Folge setzt sich fort, bis die Terminierungsbedingung greift. Kerngedanke dabei ist, dass sich die Größe des benötigten TempStores kontinuierlich verringert, da der Zeitaufwand für die Migration der Daten mit der Größe der TempStore abnimmt und somit weniger Zeit für Änderungen am Datenbestand vergeht. Die Terminierungsbedingung legt einen Schwellenwert fest, der beispielsweise die Größe des TempStores beschreibt, der während der Migration der Daten gefüllt wird. Unterschreitet nun der TempStore TSn+1 diesen Schwellenwert nach Migration des TempStores TSn, wird der gesamte Datenbestand des Altsystems für Benutzerzugriffe gesperrt und die Daten aus TSn+1 in das Neusystem übertragen. Parallel zu den genannten Aktivitäten werden spätestens in dieser Phase die Anwender für das System geschult und vorbereitet.

6. **Phase 6**: Umstellung auf neues System Die Datenbestände der beiden Systeme sind durch initiale und fortwährende Übertragung des Datenbestandes und aller Änderungen konsistent. Das Neusystem steht nun vollständig zum Einsatz bereit. Durch finales Abschalten des Altsystems und Nutzung des Neusystems wird die letzte Phase abgeschlossen und die Butterfly-Methode erfolgreich beendet.

Die Vorteile des Butterfly-Ansatzes liegen zum einen in der durchgängigen Verfügbarkeit des Altsystems. Bis auf den Zeitraum des Umschaltens, der durch intelligente Wahl des Schwellenwertes minimiert werden kann, ist das System ständig einsatzbereit und ermöglicht so die tägliche Arbeit an den Daten. Zum anderen kann zu jedem beliebigen Zeitpunkt vor dem Umschalten auf das neue System der Migrationsprozess abgebrochen werden, da die Migration umkehrbar ist, solange die Daten über die Datenzugriffskomponente gelaufen sind. Für den Fall eines Abbruchs der Migration müssen lediglich alle TempStores nacheinander in den ursprünglichen Datenbestand des Altsystems eingebunden und die Datenzugriffskomponente entfernt oder außer Kraft gesetzt werden. Je nach Datenaufkommen und Aktivität im Altsystem

können jedoch sehr viele und große TempStore notwendig sein Wu u. a. (1997). Im Extremfall führt das zu einem sehr hohen Speicherbedarf und Ressourcenverbrauch. Die Entwicklung der Datenzugriffskomponente stellt ein weiteres Problem dar.

Im Butterfly Ansatz werden im Gegensatz zum Chicken-Little-Ansatz keine Gateways benötigt, die Datenzugriffskomponente kann aber einen sehr hohen Entwicklungsaufwand erfordern. Die vorgestellten Strategien zur Systemablösung stellen durchaus unterschiedliche Möglichkeiten dar, ein System durch ein anderes zu ersetzen. Es lassen sich dabei grundsätzlich zwei Ansätze unterscheiden, das stufenweise Vorgehen und der Übergang in einem einzigen Schritt. Bei allen Strategien wird von einer gleichzeitigen Software-Entwicklung ausgegangen. Der Butterfly-Ansatz hat auch den Vorteil gegenüber dem Cold-Turkey-Ansatz, dass das Altsystem nur sehr kurz abgeschaltet werden muss, um die TempStores einzurichten. Gegenüber dem Chicken-Little-Ansatz hat er den Vorteil, dass die Integration jederzeit gestoppt werden kann, bevor das Altsystem abgeschaltet wird [STEIN08]. Dies ist beim Chicken-Little-Ansatz nicht möglich, da die Daten teilweise nur im Neusystem und teilweise nur im Altsystem vorhanden sind. Es müsste die bisherige Integration rückgängig gemacht werden. Für diese Arbeit wird der Cold-Turkey-Ansatz bevorzugt, da die Altsysteme nicht komplex sind und durch das Wrapping der alten Schnittstellen in SOA eingeführt werden.

Bei der Einführung von SOA werden die Funktionalitäten einzeln ausgelagert und als Services angeboten. Das spricht auch dafür, dass der Cold-Turkey-Ansatz verwendet wird, denn dieser ermöglicht eine in mehreren Schritten geplante Datenmigration. Außerdem ist der Cold-Turkey-Ansatz am besten geeignet für kleine Migrationsprozesse.

Die Literaturansätze stellen ein Gerüst vor, das man in der Praxis einsetzen kann. Es ist aber nur dann hilfreich, wenn die unterschiedlichen Schritte erklärt werden, die verfolgt werden müssen, um die Integration von Legacy-Systemen in Services, bzw. in einer SOA-Architecture zu schaffen. Wegen unerwarteter Probleme bei den Praxisumsetzungen weichen meistens die Praxisansätze sehr stark von der Theorie ab, daher sind die Praxisansätze praktischer und realistischer als die Theorieansätze. Im Folgenden Abschnitt werden die Praxisansätze vorgestellt mit den einzelnen Schritten, die bei der Integration einer SOA-Architecture benötigt werden.

3.2 Ansätze aus dem Praxis

Praxisansätze helfen bei der Identifikation von Zielen, Potenzialen und Herausforderungen der Integration von SOA in der Praxis. Sie zeigen konkrete Ausprägungen des SOA-Schichtenmodells (Kaplitel 2.1.2) und geben Hinweise für Maßnahmen zur Umsetzung einer SOA. Die folgenden Abschnitte untersuchen die zwei bekanntesten Praxisbeispiele einer SOA-Umsetzung.

3.2.1 SOA-Einsatz bei T-Com

Wie die meisten großen Unternehmen besitzt T-Com eine Menge von Legacy Systemen, da die IT-Architektur von T-Com durch die vollzogenen organisatorischen und technischen Entwicklungen historisch gewachsen ist.

Ausgangspunkt bildete ein hostbasiertes Anwendungssystem für die T-Com Produkte (Netzzugangsleistungen, z.B. Telefon- und ISDN-Anschlüsse). Dieses System unterstützt den kompletten Auftragserfassungs- und Produktionsprozess. Dazu werden auch die vertriebliche, vertragliche und technische Logik und Daten geregelt. Im Laufe der Zeit sind viele Erweiterungen auf der Architekture-Ebene durchgeführt, dabei entstanden z. B. für den Vertrieb von Produkten an unterschiedliche Kundensegmente (Geschäftskunden, Massenmarkt, Reseller, andere Konzerneinheiten) und über unterschiedliche Kanäle (Call Center, Internet, T- Punkte) mehrere Vertriebssysteme. Dazu wurde die Auftragsprüf- und Abwicklungslogik, die über verschiedene Schnittstellen auf die Produktionssysteme zugriff, dupliziert. Dadurch wurde der Aufwand für den IT-Betrieb erhöht und die unternehmerische Agilität reduziert. Dazu wird die Anpassung von Prozessen für die Einführung neuer Produkte komplizierter und eine selektive Auslagerung von Aktivitäten an Vertriebspartner.

T-Com ist diesen Herausforderungen begegnet und hat bereits in der Vergangenheit Anstrengungen unternommen, ihnen gerecht zu werden. Dabei wurden verschiedene Prozesse, technische oder vertragliche Fachfunktionen aus den monolithischen Anwendungssystemen herausgelöst und Schnittstellen über eine EAI-Infrastruktur konsolidiert. Gewisse Herausforderungen blieben aber trotzdem bestehen. Dadurch entstand die Notwendigkeit, eine strukturierte SOA-Architecture einzuführen. Bei der Einführung von SOA konzentrierte sich die T-Com in einem ersten Schritt auf den Prozessbereich Fulfillment (Kundenauftragsabwicklung). Zur Umsetzung der SOA waren viele Schritte notwendig, angefangen mit dem Aufbau einer SOA-Organisationsstruktur und der Definition technischer und fachlicher Architekturrichtlinien für das Servicedesign. Danach wareine standardisierte Integrationsarchitektur zu entwickeln. Das Hauptziel dieser Integrationsarchitektur besteht drin, eine Harmonisierung von Integrationsplattformen und -technologien und die Verbesserung der technischen Konnektivität zwischen Applikationen zu erreichen. Zuletzt wird die Applikationsarchitektur analysiert, um die Modularisierung und Standardisierung fachlicher Logik und Daten und die Implementierung von Services für die Interaktion zwischen Anwendungsdomänen zu schaffen. Im Folgenden werden die einzelnen Schritte genauer erklärt.

Organisationsstrukturen und Architekturrichtlinien für das Servicedesign

Neue organisatorische Rollen, Aufgaben und Verantwortlichkeiten festzulegen haben zum Ziel, die SOA im Management und in den IT-Projekten zu integrieren. Dazu sind standardisierte Prozesse für die Service-Entwicklung und -Nutzung in Fachprojekten zu definieren.

Diese beschreiben u.a., wie bei der Anforderungsanalyse, der Servicespezifikation, der Definition von Service Level Agreements (SLAs), der Service-Zertifizierung und der Registrierung von Services im zentralen Repository vorzugehen ist. Daneben erarbeiten sie eine Reihe von Architekturrichtlinien und Service-Designprinzipien. Diese umfassen technische Eigenschaften von Services (z.B. die Kommunakation zwischen Services nur über den zentralen Servicebus), fachliche Eigenschaften (z.B. Autonomität des Services und einen Mehrwert aus fachlicher Sicht liefern) sowie die Servicedokumentation. Damit diese Richtlinien und Designprinzipien in den Projekten konsistent berücksichtigt werden, entwickelt das SOA-Solution Center Templates, Werkzeuge und Methoden. Abb. 3.5 gibt z.B. einen Überblick über das einheitliche Servicebeschreibungs-Modell in einem zentralen Service-Repository.

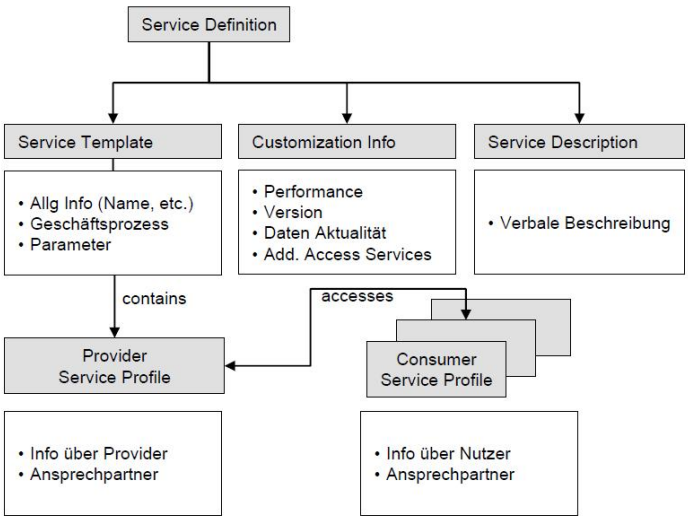

Abbildung 3.5: Servicebeschreibung im zentralen Repository (Quelle: T-Com)

Entwicklung einer standardisierten Integrationsarchitektur

Bei der serviceorientierten Zielarchitektur von T-Com werden drei Arten von Systemen unterschieden (s. Abbildung 3.6):

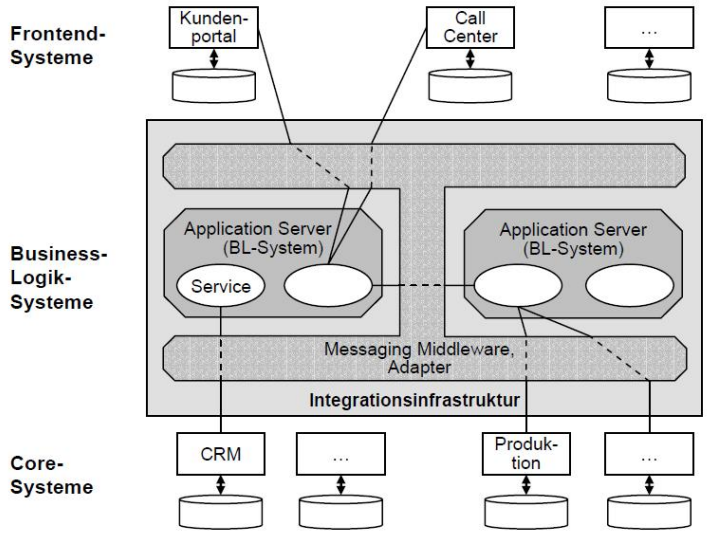

Abbildung 3.6: Elemente der SOA-Integrationsinfrastruktur

- Frontend-Systeme (z.B. Internet-Vertriebsportale), die die Schnittstellen zu Nutzern (Kunden, Call Center-Mitarbeitern etc.) representieren, dazu bieten diese Systeme kanal- und gerätespezifische Oberflächen für eine direkte Interaktion mit Endkunden bzw. Schnittstellen für die Interaktion mit T-Com-externen Anwendungssystemen (z.B. zur Kopplung von Auftragssystemen der Wholesale- Kunden). Eine weitere Aufgabe von den Frontend-Systemen besteht drin, die End-zu-End-Steuerung der Fulfillmentprozesse von der Auftragsspezifikation bis zur Auftragsverfolgung zu unterstützen und treten als Service-Konsumenten auf.

- Backend bzw. Core-Systeme sind die Anwendungssysteme, welche die fachlichen Vertriebs- und Produktionsfunktionen implementieren und die dazu notwendigen Daten führen. Beispiele sind die Stammdatensysteme oder die Produktionssysteme zur Schaltung von Leitungen, für Telefonbucheinträge, Servicetechnikerdisposition etc.

- Die Business-Logik- (BL) Systeme spielen eine Mittlerrolle zwischen Frontend- und Backend-Systemen. Dabei abstrahieren sie Dienste der Core-Systeme fachlich und technisch und orchestrieren sie je nach Anforderungen der Frontend-Systeme zu Services. Diese Systeme bieten teilweise auch Puffer-Mechanismen (z.B. eine Zwischenspeicherung zu produzierender Aufträge) und replizieren gewisse Daten um Performance- oder Verfügbarkeitsprobleme der Core-Systeme abzufangen.

Modularisierung und Standardisierung fachlicher Logik und Daten

Um den semantischen Integrationsaufwand zwischen Anwendungssystemen auf Datenebene zu reduzieren, d.h. die Übersetzung des Datenmodells einer Anwendung, entwickelt T-Com für die in den Fulfillment-Prozessen häufig verwendete Geschäftsobjekte (z.B. Kunde, Vertrag, Produkt oder Rufnummer) ein allen Services gemeinsames, applikationsübergreifendes Datenmodell (Business Object Model, BOM). Ziel des standardisierten Datenmodells ist es, einen gemeinsamen, semantisch standardisierten „Datenbus“ zu definieren und spezifische Punkt-zu-Punkt Datentransformationen zwischen einzelnen Anwendungssystemen zu vermeiden.

3.2.2 SOA-Einsatz bei Deutsche Post Brief

Die große Komplexität der IT-Systeme bei Deutsche Post Brief, dazu die geringe Verfügbarkeit der Kerndaten (Kunden, Kosten, Auftragsinformationen etc.) und zunehmend umfang- und risikoreichere IT-Projekte führten dazu, dass neue Geschäftsanforderungen bzw. neue Funktionalitäten selten in der geforderten Zeit umgesetzt werden konnten.

Die IS-Architektur wurde in der Vergangenheit nicht zentral geplant. Die Projekte aus den Fachbereichen wurden aber in Zusammenarbeit mit einer zentralen IT umgesetzt, dabei fehlte jedoch eine projektübergreifende Planung und Koordination. Dadurch sind spezialisierte, isolierte Anwendungssysteme mit großen Redundanzen in Applikationsfunktionen und Geschäftsdaten entstanden. Für Einzelprojekte wurden heterogene Schnittstellen mit mehreren undokumentierten Abhängigkeiten implementiert. Die Folgen davon waren große Anpassungsaufwände bei neuen Projekten und eine mangelhafte Wiederverwendung bestehender Funktionalitäten. Die Wartungskosten der gewachsenen Architecture stiegen dramatisch. Aus den genannten Gründen hat sich Deutsche Post Brief entschieden, ein strategisches Projekt zur zentralen, geschäftsorientierten Neugestaltung der Applikationsarchitektur zu erstellen, um eine logische Applikationsarchitektur für eine redundanzfreie Implementierung von Anwendungsfunktionalität und eine konsistente Datenpflege zu entwickeln. Ziel des Projektes war eine bessere Wiederverwendbarkeit von Funktionen zur Unterstützung neuer Prozesse sowie ein gezielter und isolierter Ausbau, um zusätzliche Funktionen zu erreichen. Daraus entstand eine fachliche SOA, bestehend aus abgegrenzter Anwendungsdomäne und Serviceschnittstelllen für den Zugriff auf Funktionen und Daten in diesen Domänen. Das strategische Projekt wurde in mehrere Teilprojekte aufgeteilt:

- Entwicklung der fachlichen Applikationsarchitektur.
- Die Implementierung einzelner Domänen und Services (z.B. der Aufbau einer zentralen Kundendatenbank).

- Entwicklung der „Service Backbone“ (SBB) und eine Integrationsplattform für die Kopplung von Serviceanbietern und Servicenutzern.

Im folgenden Abschnitt werden die Teilprojekte einzeln beschrieben:

Entwicklung der fachlichen Applikationsarchitektur

Nach einer Analyse der Kern-Geschäftsprozesse entwickelte Deutsche Post Brief eine fachliche Applikationsarchitektur, um die fachlichen Funktionen und Daten prozessunabhängig in Applikationsdomänen abzubilden (s. Abbildung 3.7). Dabei wurden die Zugriffe auf die in den Domänen gekapselten Geschäftslogik nur über Serviceschnittstellen erlaubt und die Subdomäne kommunizieren teilweise innerhalb der hauptdomänen über Services.

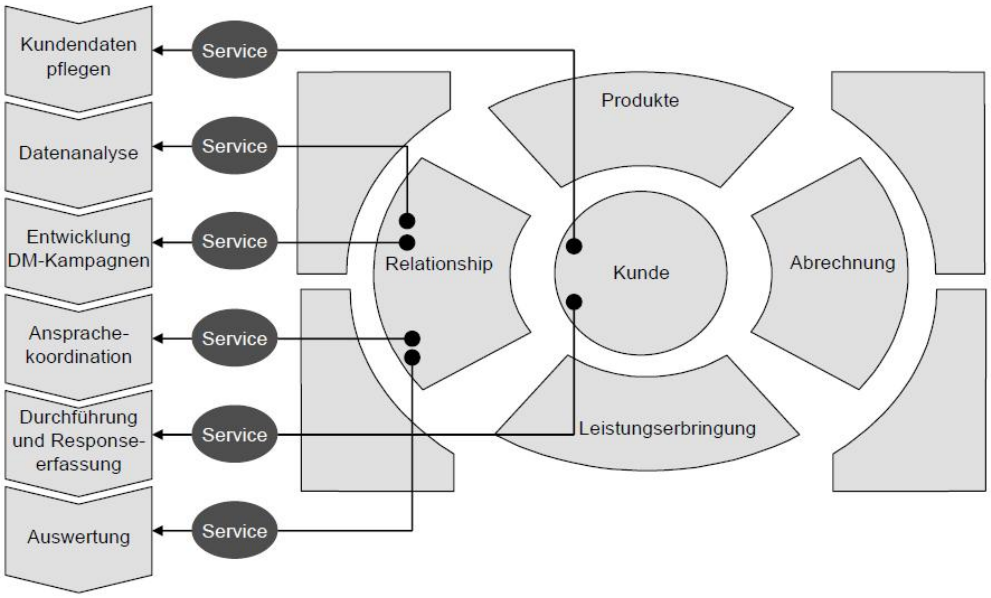

Abbildung 3.7: Applikationsdomänen und Zugriff über Services am Beispiel des Prozesses

Die identifizierten Services richten sich nach Geschäftsobjekten wie „Kunde“, „Rechnung“ oder „Vertrag“ und bieten Operationen wie „create“, „search“ oder „delete“. Wenige Services (z.B. Dublettenprüfung oder Adressvalidierung) implementieren komplexere Geschäftslogik, die sich an Prozessaktivitäten orientiert.

Identifikation und Implementierung von Services

Parallel zur Domänenbildung wurden die wichtigsten zu implementierenden Services pro Domäne identifiziert. Die Hauptkriterien, nach denen Services gebildet wurden, waren:

- Wiederverwendung: Services werden nicht nur für einen spezifischen Service-Nutzer ausgelegt, sondern möglischst für unterschiedliche Prozesse und Applikationen.
- Fachlich beschreibbare Dienstleistung: Die Services kapseln die fachliche Geschäftslogik, welche einen Geschäftsnutzen bietet, und sind aus fachlicher Sicht beschreibbar.
- Stabilität: Die Serviceschnittstellen sollten möglichst stabil sein. Dafür wurden Services entlang der grundlegenden stabilen Geschäftsfunktionen und Geschäftsobjekte definiert. Aus technischer Sicht sollen die Services eine stabile Abstraktionsschicht zwischen den Serviceanbieter und den Servicenutzern darstellen um eine schnelle und einfache Änderung der Serviceimplementierung zu ermöglischen. Um die technische Stabilität zu gewährleisten wurden die bereits die akzeptierte Industriestandards (UML, XML, WSDL etc.) für die Servicespezifikation und -entwicklung verwendet.

Bei der Serviceimplementierung legte das Unternehmen die Priorität auf Services, die zur Unterstützung neuerer Anwendungsbereiche (z.B. Customer Care, Call Center und Multi Channel Management) notwendig waren. Mit der Implementierung von Services ging teilweise auch eine Harmonisierung der Anwendungssystemlandschaft um Komplexität und Wartungs- bzw. Betriebsaufwände zu reduzieren, verfolgt die IT besonders bei Neuentwicklungen das Ziel, die Servicefunktionalität physisch möglichst in einem zentralen System (z.B. einer zentralen Kundendatenbank) zu realisieren. Dies ist jedoch nicht immer möglich, z.B. wenn für den Service benötigte Daten und Funktionen über mehrere Altanwendungen verteilt sind.

Ein Beispiel für einen implementierten Service stellt der Service „Kundenmanagement“ dar. Dieser bietet Nutzern wie bspw. Call-Center-, Portal- oder CRMSystemen über die Operationen wie z. B. „get“, „create“.. die Möglichkeit, Kundendaten zu suchen, erfassen, modifizieren und validieren. Die Servicenutzer bearbeiten die Entität Kunde über diese Operationen nicht auf Basis einzelner Attribute (Name, Vorname, Adresse etc.), wie dies bei einem Zugriff in Form von Objektmethoden typischerweise der Fall wäre, sondern erhalten und liefern jeweils eine strukturierte XMLNachricht, welche sämtliche Kundeninformationen (ca. 100 Attribute) enthält. Dadurch steigt zwar die bei der Servicenutzung ausgetauschte Datenmenge, die Anzahl Operationen bzw. Schnittstellen und die Häufigkeit der Interaktion zwischen Serviceanbieter und -nutzer wird aber niedrig gehalten. Abb. 3.8 zeigt schematisch die Anpassungen der bestehenden Anwendungssystemlandschaft bei der Umsetzung des Services: Existierten früher über 40 unterschiedliche Anwendungen mit lokaler Datenhaltung, wurde für den Service eine neue, zentrale Stammdatenbank implementiert. Diese setzt hauptsächlich Datenzugriffslogik um, mit

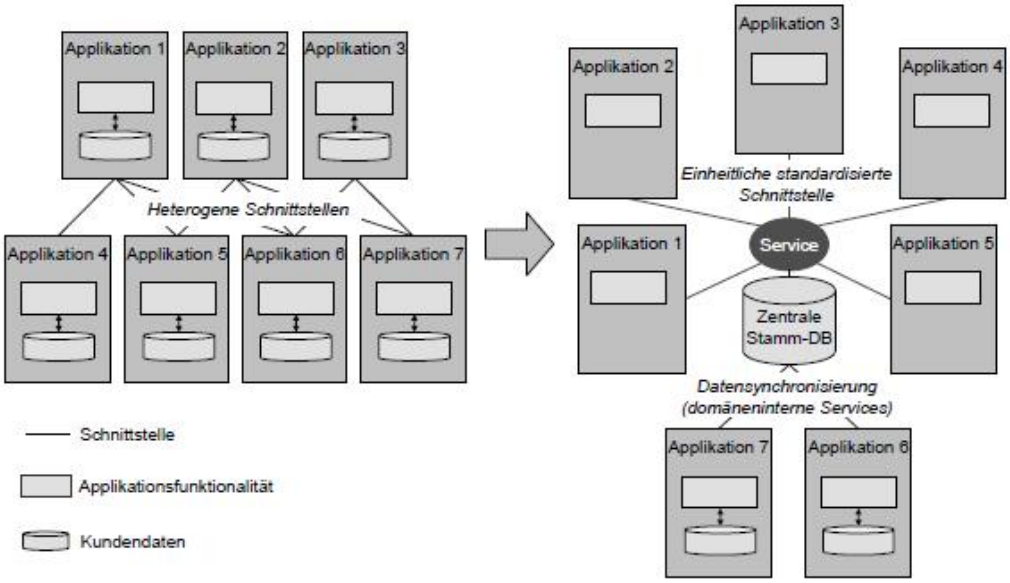

Abbildung 3.8: Implementierung des Services Kundenmanagement

nur geringem Anteil an Fachlogik bspw. für die Identifikation von Dubletten. Verschiedene bereits existierende Applikationen, besonders aber neue Applikationen, die Kundendaten verwenden, nutzen das Stammdatensystem über die Serviceschnittstelle. Derzeit existieren rund ein Dutzend Servicenutzer. Gewisse Applikationen (z.B. mobile Anwendungen ohne dauerhaften Online-Zugang) arbeiten noch mit lokalen Kundendaten, die sie aber regelmässig mit der zentralen Datenbank synchronisieren. Sie verwenden dazu spezielle, domäneninterne Serviceschnittstellen.

Entwicklung der Integrationsarchitektur (Service Backbone)

Der sog. „Service Backbone“ (SBB) stellt den Kern der technischen Umsetzung der SOA bei Deutsche Post Brief dar und unterstützt als Kernfunktion die Kommunikation zwischen Serviceanbietern und -nutzern. Die Hauptkomponenten des SBB lassen sich in die in Abb. 3.9 dargestellten Bereiche der zentralen Infrastruktur (Core), der technischen Serviceteilnehmer (Technical Service Participants), der Schnittstellen (Interface) sowie der unterstützenden Werkzeuge (Tools) für Anwendungsentwickler unterteilen. Das Service-Konzept des SBB der Deutschen Post ist mit dem Konzept der Webservices vergleichbar. Hierbei werden Webtechnologien wie SOAP und HTML mit einem zuverlässigen Massaging und einem JMS, der auf MOM basiert, verbunden.

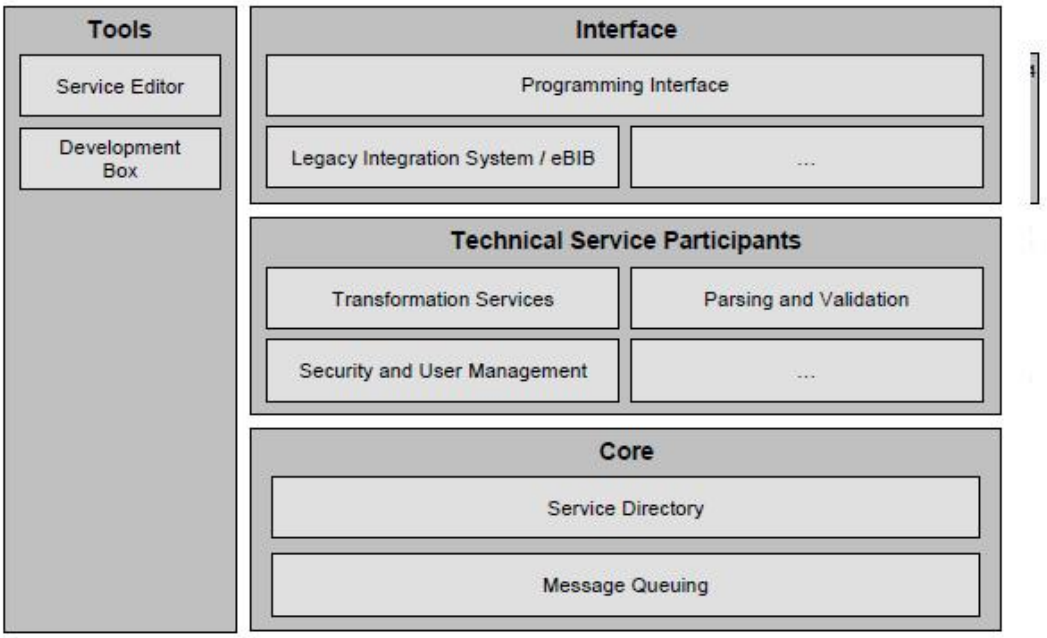

Abbildung 3.9: Komponenten des SBB (in Weiterentwicklung) [SANNEMANN04]

Der Service Backbone besteht aus folgenden drei Komponenten:

- Lokale SBB-Schnittstelle
- Zentrale Infrastrukturkomponenten
- Technische Service-Teilnehmer

In jedem Service-Teilnehmer sind lokale SBB-Schnittstellen implementiert, die eine Verbindung ermöglichen. Es gibt zwei Arten solcher Schnittstellen, Service-Provider nutzen ein SPI (Service Provider Interface), Service-Consumer ein API. Wenn ein Consumer einen Service aufruft, verwendet er das API und sendet eine Nachricht, die die Eingabeparameter enthält. Diese Nachricht wird im XMLFormat über das SBB an das SPI des Providers geschickt und die Operation wird durchgeführt.

Für die Verarbeitung eines Service-Aufrufs stehen zwei Infrastruktur- Komponenten zu Verfügung: Zum einen die Service-Registry zum anderen eine Nachrichtenwarteschlange.

- **Die Service-Registry:**

Die Registry basiert auf LDAP (Leightway Directory Access Protocol) und wird mit UDDI (Universal Description, Discovery and Integration) entwickelt. Um höchste Verfügbarkeit zu gewährleisten, werden diese Infrastrukturkomponenten abgeglichen und geclustert.

- **Nachrichtenwarteschlange:**

Eine Nachrichtenwarteschlange ist ein Puffer, der Nachrichten abfängt und sie in die Warteschlange einfügt. Sie werden weitergeleitet, sobald neue Kapazitäten frei werden.

Ein Service-Aufruf besteht hauptsächlich aus einem XML-Dokument mit den Attributen der Geschäftsobjekte, die erzeugt, manipuliert oder verändert werden sollen. Zusätzliche Parameter können genutzt werden, um beispielsweise synchrone oder asynchrone Kommunikation zu steuern.

Die Service-Registry ist die zentrale Quelle für die Kommunikation zwischen Interaktionspartnern. Um welche Interaktion es sich dabei handelt ist dabei von zweitrangiger Bedeutung, da der Interaktionstyp in einem eigenen Parameter übergeben wird. Die Schnittstellen greifen dabei immer auf das Service-Backbone zu. Es gibt die Kommunikationsarten synchron/asynchron, Publish/Subscribe und Frage/Antwort. Bevor das Service-Backbone realisiert wurde, basierte die IT-Strategie der deutschen Post auf C++. Daher war es nötig, die neue SOA so offen zu gestalten, dass, wenn nötig, auch Argumente in C++-Form verschickt werden können. Der Aufruf des SBB ist ein Java-Aufruf, wobei das Hauptargument ein XML-Dokument ist. Die Struktur der XML-Dokumente wird über die servicespezifischen XML-Schemata definiert. Intern wird SOAP mit Java verwendet, wobei es Wrapper und Adapter für C++- bzw. Nicht-XMLArgumente gibt. Das Service Backbone selbst basiert auf loser Kopplung, verwendet Standards und vermeidet proprietäre Merkmale. Obwohl MQSeries bei der Nachrichtenverarbeitung eingesetzt wird, werden keine proprietären Funktionen verwendet, da die Verbindung zu MQSeries durch eine JMS-Schnittstelle realisiert wird. So wird eine leichtere Austauschbarkeit gewährleistet und verhindert, dass durch die Verwendung von nicht-standardisierten Merkmalen eine Anbieterabhängigkeit entsteht. Ein ähnlicher Ansatz wird auf für andere Komponenten verfolgt. Die technischen Service-Teilnehmer der Deutschen Post zum vorgegebenen Zeitpunkt waren Transformation, Service Registry Administration, Data Integration und Single Sign On.

Durch Verwendung des SBB bleiben Implementierungsprojekte überschaubar und schlank. Diese Projekte können sich auf die Service-Implementierungen konzentrieren und müssen sich nicht um Datenanbindung kümmern. Darüber hinaus kann der SBB verwaltet und aktualisiert werden, ohne dass die eigentlichen Services verändert werden müssen. SOA und SBB haben sich bei der Deutschen Post trotz einiger Probleme in der Anfangsphase als sehr erfolgreich erwiesen und wurden auch später ständig erweitert, indem aktuelle und potentielle Benutzer in den Gesamtprozess miteinbezogen wurden.

3.3 Zusammenfassung

Bei der Untersuchung der Theorieansätze wurden zuerst die Integrationsmöglichkeiten vor gestellt. Es existieren hauptsächlich drei unterschiedliche Integrationsansätze. Bei dem ersten Ansatz wird die existierende Applikation Infrastruktur beibehalten und nur die Schnittstelle als Webervices entwickelt (Kapitel 3.1.1). Beim zweiten Anzatz werden alle Legacy-Systeme neu entwickelt und in Services integriert (Kapitel 3.1.1). Der dritte Ansatz ist eine Mischung von den zwei ersten Ansätze, dabei werden nur Teilkomponenten des Legacy-Systems neu entwickelt (Kapitel 3.1.1).

Beim Wrapping Ansatz werden nur die Schnittstellen neu entwickelt, dadurch fordert dieser Ansatz keine genaue Analyse des Gesamtsystems und die gesamten Funktionalitäten werden beibehalten. Der Nachteil dabei ist die schwere Wartbarkeit des Systems, denn die einzelnen Funktionen werden als Blackboxes dargestellt. Die Möglichkeit, nur Teile aus den Legacy-Systemen neu zu entwickeln ist meistens die plausible Lösung und wird favorisiert. Dabei werden die Vorteile aus den zwei ersten Ansätze übernommen.

Bei der Neuentwicklung aller Komponenten oder der Teilkomponenten des Legacy-Systems stellt sich meistens die Frage, ob eine Migration der Daten stattfinden muss, denn die Datenübernahme nimmt eine gewisse Zeit im Anspruch und die Verfügbarkeit des Altsystem wird beeinträchtigt. Deshalb wurden die möglichen Migrationsansätze vorgestellt, die für die Verfügbarkeit des Altsystems, während der Datenübernahme im Neusystem sorgen. Im Allgemeinen existieren drei bekannte Migrationsansätze. Der erste Ansatz, der „Cold-Turkey-Ansatz“, versucht das Altsystem weiterhin zu nutzen, während das Neusystem entwickelt wird.

Beim zweiten Ansatz, dem „Chicken-Little-Ansatz“, wird das Altsystem Stück für Stück durch das neue System abgelöst unter Verwendung von Gateways. Den dritten Ansatz, den „Butterfly-Ansatz“, kann man als reine Datenmigration betrachten, dabei wird das Neusystem in einer Testumgebung entwickelt, ohne den alltäglichen Systembetrieb der Altanwendung zu beeinflussen oder zu stören.

Der Butterfly Ansatz hat auch den Vorteil gegenüber dem Cold-Turkey-Ansatz, dass das Altsystem nur sehr kurz abgeschaltet werden muss, um die TempStores einzurichten. Gegenüber dem Chicken-Little-Ansatz hat er den Vorteil, dass die Integration jederzeit gestoppt werden kann, bevor das Altsystem abgeschaltet wird [STEIN08]. Dies ist beim Chicken-Little-Ansatz nicht möglich, da die Daten teilweise nur im Neusystem und teilweise nur im Altsystem vorhanden sind. Es müsste die bisherige Integration rückgängig gemacht werden. Für diese Arbeit wird der Cold-Turkey-Ansatz bevorzugt, da die Altsysteme nicht komplex sind und durch das Wrapping der alten Schnittstellen in SOA eingeführt werden. Bei der Einführung von SOA, werden die Funktionalitäten einzeln ausgelagert und als Services angeboten. Das spricht auch dafür, dass der Cold-Turkey-Ansatz verwendet wird, denn dieser ermöglicht eine in mehreren

Schritten geplante Datenmigration. Außerdem ist der Cold-Turkey-Ansatz am besten geeignet für kleine Migrationsprozesse.

Die Literaturansätze stellen ein Gerüst vor, das man in der Praxis einsetzen kann. Die Untersuchung der Praxisansätze zeigt die Auslöser für die Einführung von SOA, um die Integrationsaufgaben durchzuführen. Dabei waren folgende Punkte die wichtigsten Auslöser dieser Integration:

- Mangelhaft integrierte Geschäftsprozesse
- Lange und risikoreiche Projekte beim Umsetzen neuer Geschäftsanforderungen
- Hohe Kosten des IT-Betriebs
- und Probleme bei notwendigen technischen Modernisierungen oder Migration des altsystems in der IS-Architektur.

Die Beispiele zeigen, dass die Umsetzung von SOA erst in den Anfängen steckt. Keines der Unternehmen realisiert sämtliche Architekturebenen des SOA-Modells vollständig. Dabei legten beide Praxisbeispiele das Hauptgewicht auf die Entwicklung einer Domänenarchitektur sowie die Umsetzung standardisierter, gemanagter Serviceschnittstellen auf einer zentralen Integrationsinfrastruktur. Sie bewegen sich damit primär auf der Anwendungssystem- und der Service-Ebene. Die Logik zur Komposition von Services wird überwiegend direkt in die nutzenden Applikationen programmiert.

Die Fallstudien zeigten die drei Hauptziele von SOA: standardisierte Integrationsinfrastruktur, Entkoppelung von Appliktionsdomänen und flexible Benutzer- bzw. Geschäftsprozessintegration. Die SOA-Ebenen und die Designprinzipien werden je nach Ausgangslage und Zielsetzung unterschiedlich gewichtet. Obwohl mehrere Unternehmen bereits seit mehreren Jahren an der Umsetzung der SOA arbeiten, hat keines sämtliche Ebenen ausgeprägt. Insbesodere die flexible Komposition und Orchestrierung von Services auf der Ebene Workflow- und Desktopintegration - in der Literatur oft als eines der Hauptpotenziale von SOA aufgeführt - wollen die meisten Unternehmen erst in Zukunft angehen. Im Rahmen dieser Arbeit wird dieser Aspekt nicht betrachtet. Die Fallstudien liefern auch Hinweise dafür, welche Maßnahmen für die Umsetzung einer SOA zu treffen sind, was die Herausforderungen sind und welche Handlungsoptionen bestehen.

4 Analyse

Das Kapitel wird zuerst durch ein mögliches Anwendungs-Szenario motiviert. Daraus werden die Anforderungen für die neue Architektur extrahiert. Hiernach wird der Umfang der Zielarchitektur festgelegt. Desweiteren wird die aktuelle Architektur des E-Commerce-Unternehmens vorgestellt, die Geschäftsprozesse werden dargestellt und analysiert. Daraus werden die technischen Ziele der Arbeit als präzise Aufgabenstellung formuliert. Das Ziel dieser Arbeit, ist zu zeigen, wie sich durch Einsatz einer neuen Architektur für lose gekoppelte Systeme vorhandene Funktionalität für unterschiedliche Vertriebswegen nutzen lässt.

Ein weiteres Ziel dieser Arbeit besteht darin, die Vorgehensweise zu definieren, wie die neue Architektur realisiert werden kann und wie die Komponenten aus dem alten System integriert werden können. Im zweiten Abschnitt dieses Kapitels werden die funktionalen sowie die nicht funktionalen Anforderungen formuliert, damit die entsprechenden technischen Umsetzungsmöglichkeiten einer SOA vorgestellt werden können.

4.1 Anwendungs-Szenario

Die meisten E-Commerce Systeme verfügen über ein Shop-System mit einem oder mehreren Vertriebswegen, welches an eine ERP oder ein Fullfilment-System angebunden ist. Es besteht meistens der Bedarf, im Falle der Fakturierung oder des Ausfalls vom ERP-System oder vom Fullfilment System über eine zusätzliche Applikation (Callagent) Kundenaufträge zu erfassen.

Callagent ist eine Applikation, die es Mitarbeitern des Callcenters ermöglicht, für die Zeit der Fakturierung oder des Ausfalls vom ERP-System oder vom Fullfilment System webbasiert Kundenaufträge zu erfassen. Die Applikation soll dabei möglichst die gleichen Schnittstellen und Funktionalitäten des bereits entwickelten Shopsystems besitzen.

Im Folgenden werden die Anforderungen des Callagents aufgelistet:

- Eine Anmeldung des Callcentermitarbeiters pro Vertriebsweg

- Der Vertriebsweg muss für den Erfasser jederzeit deutlich zu erkennen sein
- Der Erfasser muss nachvollziehbar sein
- Der Erfasser muss Kunden suchen und anlegen können (Lucene)
- Der Erfasser muss nach Produkten suchen können (Factfinder)
- Der Erfasser muss die Adressen-Daten validieren können (Fuzzy)
- Der Erfasser muss die Kundenbonität prüfen können (Creditreform)
- Der Erfasser muss einen Warenkorb anlegen können
- Der Erfasser muss eine Bestellung anlegen können
- Der Erfasser muss eine Bestellung bezahlen können
- Der Datenschutz muss gewährleistet sein
- Eine sichere Verbindung muss gewährleistet sein

Die beste technische Umsetzung der Applikation könnte mit denselben Komponenten erfolgen, die vom Shop-System benutzt werden, um eine parallele Verwendung der bestehenden Systeme zu garantieren. Ähnlich wie das Shop-System soll das CallAgent-System im ersten Schritt die Produktsuche ermöglichen (Abbildung 4.1). Im zweiten Schritt soll die Bestellmenge überprüft werden. Danach findet die Kundendaten-Eingabe statt und die Adressenvalidierung. Im letzte Schritt wird nach der Zahlungsart-Auswahl die Bonität des Kunden geprüft.

Im Folgenden findet eine Beschreibung der dargestellten Funktionalitäten statt:

- Produktsuche Die Suche nach Produkten wird primär mittels Bestellnummern erfolgen. Die Erfasser orientieren sich an den Aussagen der Kunden und mittels Printkatalog.

  Bei erfolgloser Suche kann auf die Funktionalitäten der FactFinder-Suche zurückgegriffen werden.

- Bestand Prüfen: Der Bestand wird täglich vom Fullfilment System ermittelt und bei jeder Bestellung runtergezählt. Während des Bestellprozesses wird der verfügbare Bestand mehrmals danach geprüft, ob die zu bestellende Menge noch verfügbar ist.

- Adressenvalidierung: Beide Validierungsdienste AdressDoctor und Fuzzy werden als Extra-Komponente angelegt und für beide Systeme zur Verfügung gestellt, für das Shop-System sowie für das Callagent

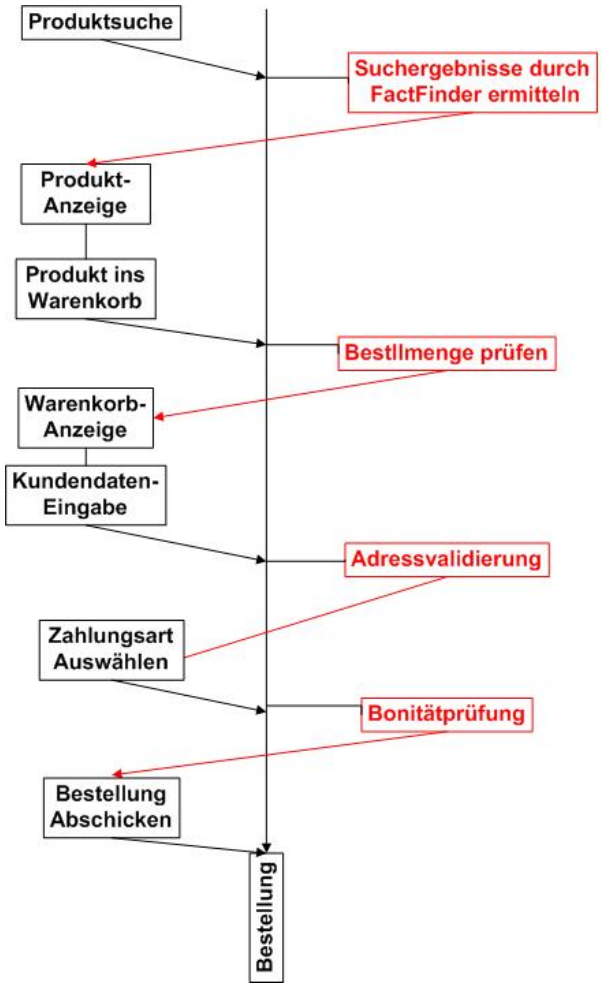

Abbildung 4.1: Ablaufdiagram CallAgent Bestellung

- Bonitätsprüfung: Bei der Entscheidung welche, Zahlungsart der Kunde verwenden darf, wird eine Bonitätsprüfung bei Creditreform durchgeführt. Dafür wird eine neue Komponente angelegt, die die einzelnen Abfragen direkt bei Creditreform durchführt und das Ergebnis interpretiert.

Um das Anwendungsszenario zu realisieren ist eine vorherige Analyse des Ist-Zustandes notwendig. Im nächsten Abschnitt wird die aktuelle System-Architektur vorgestellt um die Anforderung der neuen Architektur zu definieren.

In diesem Abschnitt hat eine kurze Vorstellung des CallAgent-Systems als Anwendungsszenario stattgefunden. Dabei wurden die funktionalen Anforderungen des neuen Systems darge-

stellt. Dazu wurden die Komponenten vorgestellt, die in der aktuellen Architektur existieren und übernommen werden können. Im nächsten Abschnitt findet eine Analyse des Ist-Zustandes statt. Darauf folgt eine Auswertung, welche Komponenten vom aktuellen Shop-System getrennt werden können und welche davon beibehalten werden können.

4.2 Analyse des Ist-Zustandes

Diese Arbeit basiert auf der Architektur eines E-Commerce-Unternehmens. Das Hauptziel dieser Arbeit besteht drin, eine spezielle Vorgehensweise, bzw. Lösung für das Unternehmen zu finden, die eine zukünftige Integration neuer Komponente einfacher ermöglichen kann. Daraus soll eine allgemeine Vorgehensweise für E-Commerce-Systeme extrahiert werden. Dabei wird im Rahmen dieser Arbeit die SOA-Architektur in die Unternehmensarchitektur eingeführt, damit zukünftige Erweiterungen durch neue oder externe Komponenten einfacher werden.

Im Folgenden wird zuerst die Softwarearchitektur beschrieben (Abbildung 4.2), dazu folgt eine Vorstellung der Systemarchitektur aus funktionaler Sicht. Desweiteren werden die einzelnen E-Commerce-Prozesse der Unternehmen genauer erklärt.

4.2.1 Softwarearchitektur

Wie in der Abbildung 4.2 zu sehen ist, besteht die aktuelle Softwarearchitektur des Unternehmens aus drei Systemen, die miteinander kommunizieren und Daten austauschen.

Zwischen dem Fullfilment-System und dem ERP-System besteht keine direkte Kommunikation oder kein Austausch der Daten, sondern es laufen alle Operationen nur über das Shop-System. Zwischen dem Shop-System und dem ERP-System findet der Datenaustausch mittels File-Schnittstellen statt. Dabei werden Produktdaten vom ERP-System ans Shop-System exportiert, dazu Bestell- und Stornodaten vom Shop-System ans ERP-System exportiert.

Die externe Dienste werden direkt aus dem Shop-System aufgerufen und die Logik davon ist in der Business-Logik der Applikationseinheit implementiert(Abbildung 4.3). Ein Beispiel für eine integrierte Logik und einen Dienstaufruf in der Shop-Applikation ist der Adressvalidierungsdienst Fuzzy. Dabei wird vom Shopsystem ein request in Form von einer XML-Datei generiert, dann wird ein Request seitens Fuzzy durchgeführt. Als Responce bekommt das Shopsystem eine XML-Datei, die als JDOM dargestellt wird. Das Parsen dieser Responce und die Interpretation der Ergebnisse befindet sich im Shop-System. Die Logik der Abarbeitung von Bestellungen steckt auch im Shop-System (BestellungsManager). Dabei entscheidet das Shop-System, wann eine Bestellung an das ERP-System oder an das Fullfilment-System exportiert werden

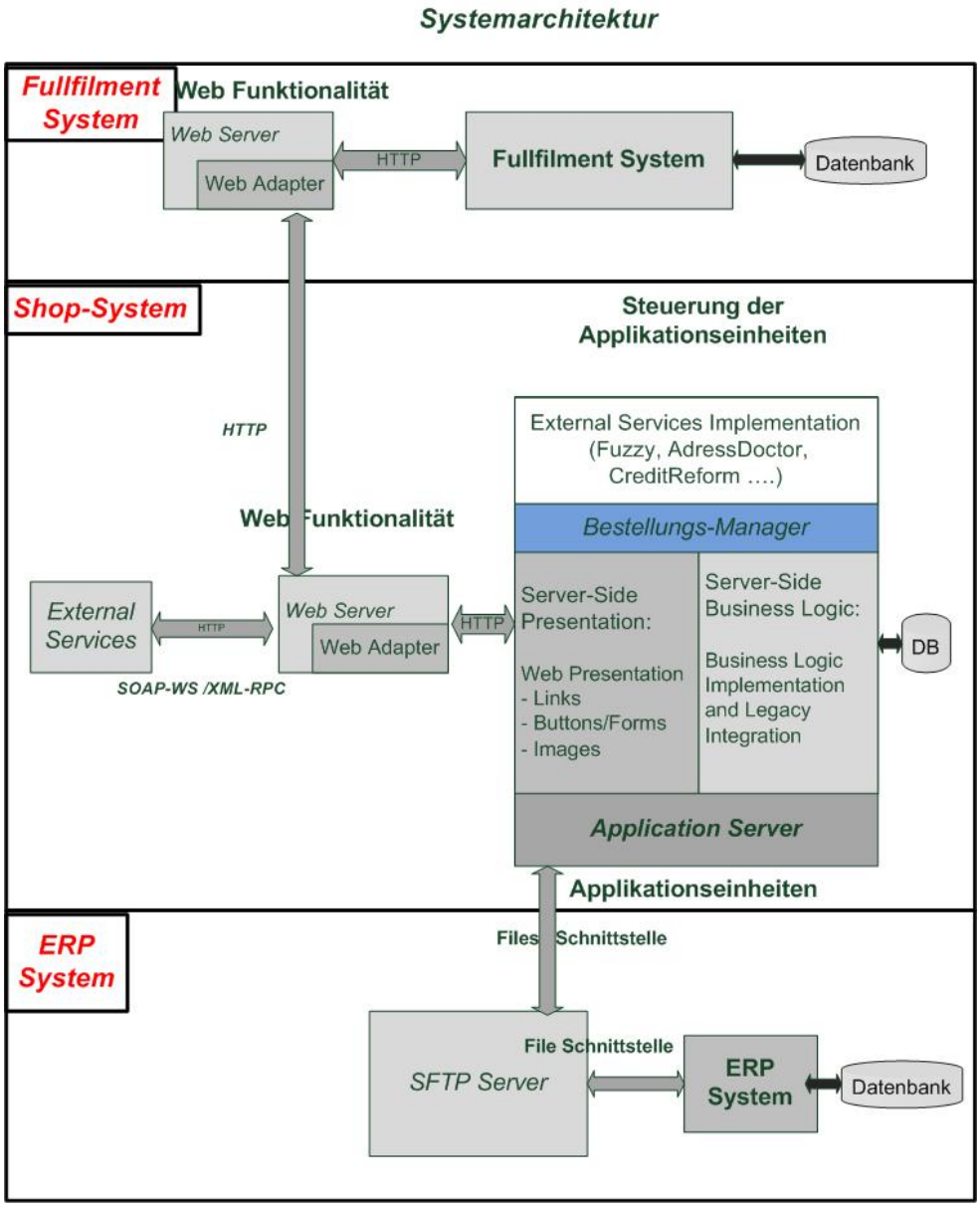

Abbildung 4.2: Softwarerchitektur des Unternehmens

soll und in welchem Format. Diese Logik kann auch als Extra-Komponente ausgelagert werden. In der zukünftigen Architektur wird das Erstellen der Request, das Parsen der Responce und die Interpretation der Ergebnisse in einer neuen Komponente umgelagert und extern ange-

boten, damit die gleiche Funktionalität nicht mehrfach im System implementiert werden muss und eine spätere Anpassung, bzw. Erweiterung mehrfach durchgeführt werden muss.

4.2.2 Systemarchitektur

Die Systemarchitektur (Abbildung 4.3) besteht aus drei Haupt-Systemen (das Shop-System, das Enterprice-Resource-Planing-System (ERP-System) und das Fullfilment-System).

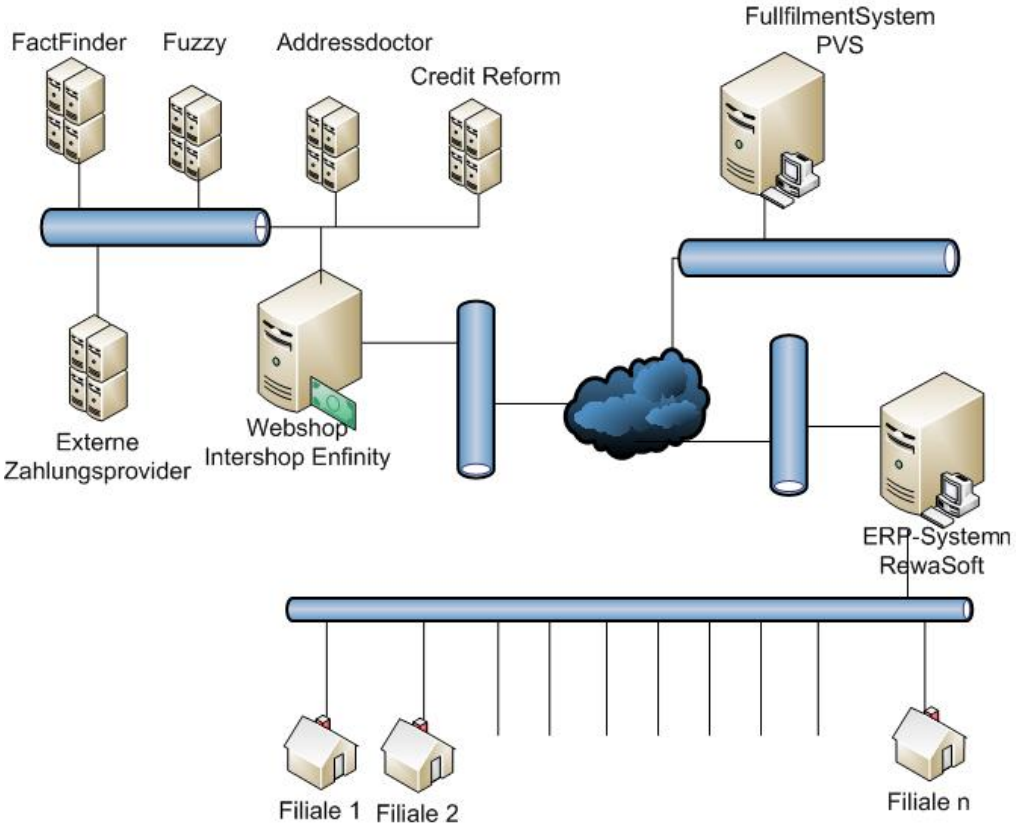

Abbildung 4.3: Systemarchitektur des Unternehmens

Im Folgenden findet eine Beschreibung der drei Systeme aus funktionaler Sicht statt:

Beschreibung Shop-System

Das Shop-System hat die Aufgabe, eine Geschäftsbeziehung zwischen und unter Anbieter und Abnehmer abzuwickeln. Dabei werden die Kunden- und Bestelldaten angenommen, geprüft und weitergegeben zwecks Verarbeitung (Abbildung 4.3).

Im Mittelpunkt dieser Architektur steht das Shop-System auf der Basis einer Standard-Shop-Systems Intershop Enfinity.

Das Shop System spielt nicht nur die Rolle eines Shopsystems, sondern auch einer Middleware, dazu versorgt das Shop System weitere Systeme mit Bestellungs-, Kunden- und teilweise mit Produktdaten.

Die Produkte werden im ERP-System angelegt, initialisiert und dann an das Shop-System mittels XML-Files exportiert. Im Shop-System werden die Produktdaten redaktionell gepflegt, bearbeitet und danach via SOAP als basierte Webservices und als eine File-Schnittstelle an das Fullfilment-System exportiert.

Das Shop-System steht auch mit externen Diensten in Verbindung, die unterschiedliche Funktionalitäten anbieten. Für die Adressenprüfung, bzw. Validierung nimmt das Shop-System die Dienste von Fuzzy und AdressDoctor in Anspruch. Dabei ist eine Logik implementiert, die bestimmt, für welche Adressen welcher Dienst verwendet werden soll. Der Austausch der Daten zwischen dem Shop-System und die Adressprüfungsdienste geschieht mittels XML-RPC und SOAP-Webservices. Für die Bonitätsprüfung wird der Service Creditreform verwendet, dabei wird für Auswahl der Zahlungsmethode Rechnung eine gute Bonität vorausgesetzt.

Externe Zahlungsprovider werden auch direkt vom Shop-System angesprochen. Für Zahlungen mit Kreditkarte wird z. B. Saferpay verwendet. Dabei erfolgt die Zahlung in zwei Schritten. Beim ersten Schritt werden die Daten an Saferpay weitergeleitet und eine Autorisierung durchgeführt. Beim zweiten Schritt wird mit Hilfe von Webservices ein „capture“ durchgeführt. Paypal und Paypal Express werden auch angeboten. Zahlungen mit OnlineBanking werden über Sofortüberweisung angeboten.

Für die Produktsuche wird Factfinder eingesetzt. Diese Anwendung bietet einen Service für Produkt-Suche, der über Webservices aufgerufen wird. Dieses Webservices wird vom Shop-System aufgerufen.

Bestellungen werden auch vom Shop-System durch den BestellungsManager abgearbeitet. Dabei fällt die Entscheidung, wann und wohin eine Bestellung exportiert bzw. importiert werden kann und in welchem Format.

Technik Die E-Commerce Plattform Enfinity baut auf Standardtechnologien und offenen Standards auf, die die Integration mit bestehenden Systemen und neue Technologien ermöglichen sollen. Technologien wie Java, EJB, JSP und XML bilden das Grundgerüst von

Enfinity, auf dem das gesamte System aufgebaut ist. Neben dem Einsatz dieser Technologien wird bei Enfinity eine Trennung zwischen Präsentationsebene, Geschäftslogikebene und Datenebene vorgenommen. Diese Trennung bei der Architektur des Systems ist an das Verteilungsmodell für Client-Server-Architekturen angelehnt.

Beschreibung Fullfilment-System

Das Fullfilment-System hat die Aufgabe Bestellungen abzuwickeln und Bestandsführung für das Shop-System durchzuführung. Dabei übernimmt das Fullfilment-System die Logistik und tauscht alle Produkt- und Bestellinformationen mit dem Shop-System über bestehende Webservices- und Files-Schnittstellen aus:

- Bestellungsservice: Dieser Service importiert die Bestellungen aus dem Shop-System mittels Webservices und File-Schnittstelle.
- Bestellungsstatusservice: Dieser Service informiert das Shop-System über jeden Fortschritt aller Bestellungen.
- Stornierungsservice: Dieser Service exportiert die stornierten Bestellungen an das Shop-System mittels Webservices und File-Schnittstelle.
- Retoureservice: Dieser Service exportiert alle Retoure an das Shop-System mittels Webservices und File-Schnittstelle.
- Produkteservice: Dieser Service importiert alle aktualisierten Produkte aus dem Shop-System mittels Webservices und Fileschnittstelle.
- Produktbestand: Dieser Service informiert das Shop-System ständig über die verfügbare Menge aller Produkte im Lager.

Beschreibung ERP-System

Das ERP-System hat die Aufgabe, die Verwaltung von Informationen zur Kundenpflege, die Erstellung von interaktiven Katalogen (Online-Shop) und die interne Bestandsführung der einzelnen Filialen zu gewährleisten. Dazu hat das ERP-System die Aufgabe, Bestellinformationen zentral zu speichern und für alle Filialen sichtbar zu machen.

Auch das ERP-System tauscht alle Bestell- und Produkt-Informationen mit dem Shop-System aus. Dabei werden folgende Services verwendet:

- Stornierungsservice: Dieser Service importiert die stornierten Bestellungen aus dem Shop-System mittels File-Schnittstelle.

- Bestellungsservice: Dieser Service importiert die Bestellungen aus dem Shop-System.
- Retoureimportservice: Dieser Service importiert alle Retoure aus dem Shop-System.
- Retoureexportservice: Dieser Service exportiert alle in den Fachmärkten retournierten Aufträge an das Shop-System.
- Produkteservice: Dieser Service exportiert alle aktualisierten Produkte an das Shop-System.

Nachdem die drei Haupt-Systeme vorgestellt wurden, werden im nächsten Abschnitt die E-Commerce-Prozesse der Unternehmen beschrieben, bzw. modelliert, damit diese Prozesse bei der Modellierung der neuen Architektur berücksichtigt werden können.

4.2.3 Prozess Beschreibung

Das Beispiel-Unternehmen verwendet seine eigene Prozess-Engine um die auszuführenden Prozesse - also die vorstrukturierte Abfolge von einzelnen Aktivitäten - zu steuern. Die Prozessengine wird im Rahmen dieser Arbeit nicht betrachtet. Momentan sind die folgende Prozesse im Beispiel-Unternehmen (Abbildung 4.4) abgebildet:

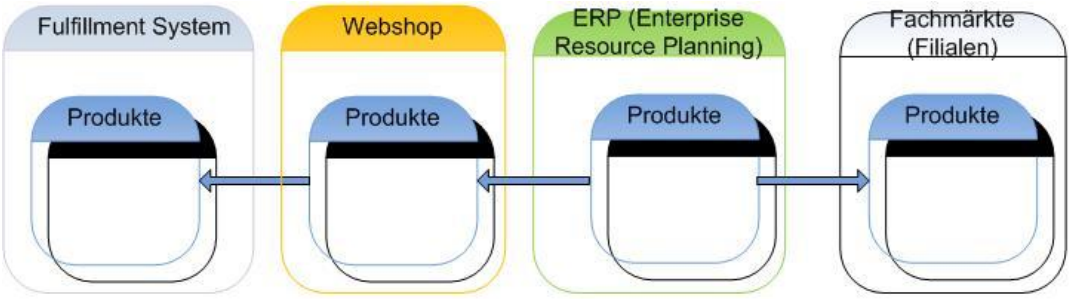

Abbildung 4.4: Prozessablauf Produkte

- **Prozess-Ablauf Produkte:** Die Produktdaten werden zuerst im ERP-System angelegt und dann zum Webshop übertragen, danach werden die Produkte im Webshop bearbeitet und anschließend an das Fullfillment-System exportiert (Abbildung 4.4) .

- **Prozess-Ablauf Bestellungen:** Eine Bestellung wird im Webshop erzeugt, danach wird die Bestellung zum ERP-System und Fullfillment-System übertragen. Die Übertragung an das Fullfillment-System hat den Zweck, die Bestellung auszuliefern aber die Übertragung an das ERP-System dient dazu, die Bestellung an alle Fachmärkte als Information zu übertragen (Abbildung 4.5) .

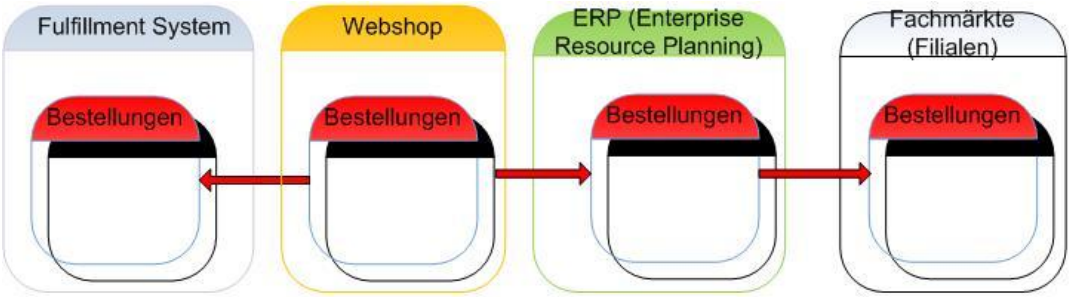

Abbildung 4.5: Prozessablauf Bestellung

- **Prozess-Ablauf Retoure:** Für den Kunden gibt es zwei Möglichkeiten, eine Bestellung zu retournieren:
  - *Retoure im Fachmarkt:* Der Kunde kann seine Bestellung direkt bei einem Fachmarkt retournieren, bzw. umtauschen. Diese Information wird zuerst an das ERP-System exportiert und danach bekommt das Shop-System diese Information vom ERP-System übertragen (Abbildung 4.6).
  - *Retoure an das Fullfillment System:* Für den Kunden gibt es die Möglichkeit, seine Bestellung über das Fullfillment-System zu retournieren. Diese Information wird zuerst an das Shop-System exportiert und danach bekommt das ERP-System diese Information vom Shop-System übertragen (Abbildung 4.6) .

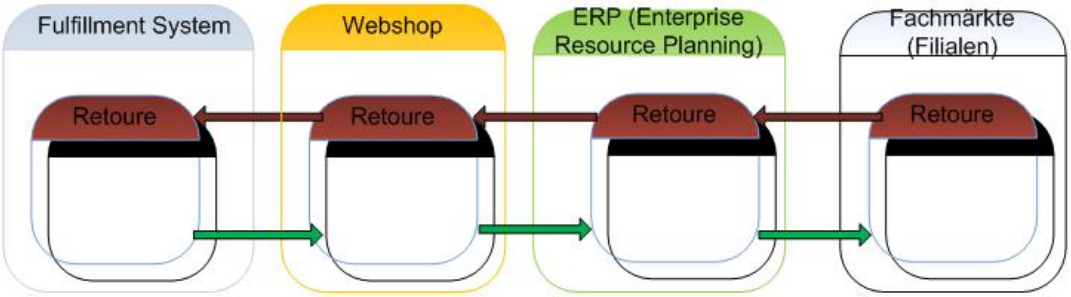

Abbildung 4.6: Prozessablauf Retoure

- **Prozess-Ablauf Stornierung:** Eine Stornierung wird im Fullfillment-System erzeugt, danach wird die Stornierung zum Shop-System und dann zum ERP-System übertragen. Mit der Übertragung an das ERP-System werden die Stornierungen automatisch an alle Fachmärkte als Information übertragen (Abbildung 4.7) .

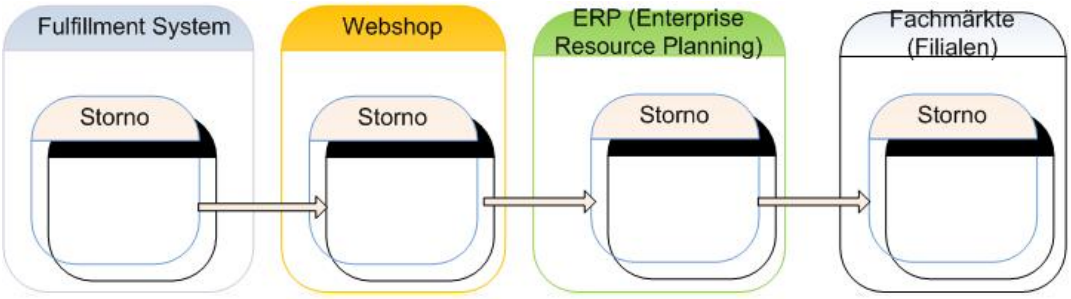

Abbildung 4.7: Prozessablauf Stornierung

Um eine Bestellung zu erzeugen werden viele Aktivitäten durchgeführt, z. B. Produkte suchen, registrieren und Bestellung bezahlen. Dieser Prozess wird in Abbildung 4.8 genauer erklärt.

Das erste Schritt für den Kunden im OnlineShop ist meistens die Produktsuche. Beim Beispielunternehmen wird Factfinder als externer Suchserver für die Suche verwendet. Als nächster Schritt kommt der Warenkorbprozess. Dabei werden die Produkte in den Warenkorb gelegt. Anschließend findet der Registrierungsprozess statt. In diesem Prozess wird die Adressenvalidierung durch Fuzzy oder AdressDoctor durchgeführt.

Nach der Registrierung wird der Kunde zur Zahlung weitergeleitet, dabei besteht die Mög lichkeit einen externen Zahlungsprovider zu verwenden (Sofortüberweisung, Paypal oder Saferpay). Sobald der Kunde seine Bestellung erfolgreich bezahlt hat, wird daraus ein Auftrag erzeugt.

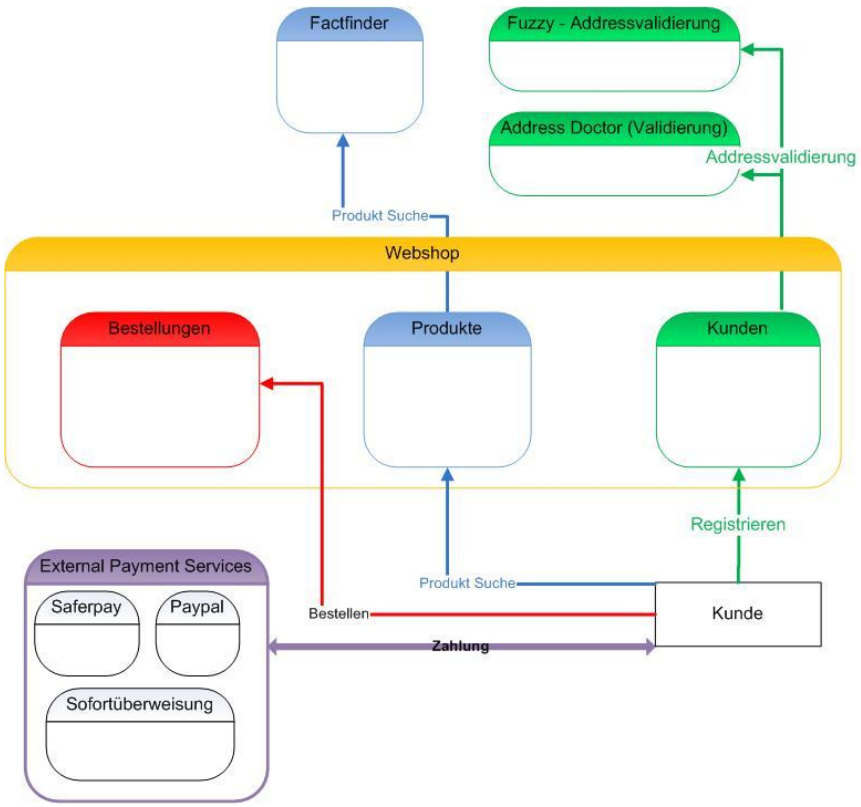

Abbildung 4.8: Geschäftsablauf Bestellung erzeugen

4.2.4 Zusammenfassung

Beim ersten Teil dieses Abschnitts wurde das Anwendungs-Szenario vorgestellt. Danach wurde ein Überblick auf die aktuelle Softwarearchitecture vorgestellt. Der zweite Abschnitt dieses Kapitels stellte die aktuelle Systemarchitektur aus funktionaler Sicht vor und die einzelnen Prozesse wurden genauer erklärt. In der aktuellen Architektur existierten Funktionalitäten, die man doppelt implementieren konnte und möglicherweise wichen diese Implementierunen voneinander ab. Dadurch könnte auch eine Integrationsproblematik entstehen.

Im folgenden Anschnitt werden zuerst die möglichen Integrationsprobleme, die in Zukunft auftreten können, definiert. Danach findet eine Aufgabenstellung mit der Analyse der funktionalen Anforderungen statt. Dabei wird das System in Komponenten zerlegt und die einzelnen Komponenten werden dann genauer definiert. Im Anschluss daran findet die Entscheidung statt, welche Komponenten im Shop-System beibehalten werden können und welche davon als externe Komponenten angeboten werden können.

4.3 Anforderungsanalyse

In diesem Abschnitt werden die technischen und konzeptionellen Anforderungen dargelegt, die bei der Realisierung der neuen Architektur erfüllt werden sollten. Im Anschluss findet eine detaillierte Spezifikation der Anwendungsfälle statt. Hier werden Use-Cases ausführlich spezifiziert. Abschließend werden die nicht funktionalen Anforderungen erläutert. Dazu zählen die technischen Anforderungen.

4.3.1 Motivation zur Lösung der Integrationsproblematik

Das E-Commerce-Unternehmen besitzt ein monolithisches Shop-System, das die Rolle eines Vermittlers spielt. Darin befindet sich das Shop-System im Mittelpunkt der Softwarerchitektur und alle externen Systeme sind direkt mit dem Shop-System verbunden. Dazu ist ein großer Teil der Businesslogik ins Shop-System implementiert. Ohne Shop-System kann kein Prozess durchgeführt werden. Für das E-Commerce-Unternehmen bedeutet das einen großen Aufwand bei zukünftigen Erweiterungen des Systems, besonders wenn ein externer Dienst mehrfach verwendet werden sollte oder eine neue Komponente integriert werden muss. Die Komplexität der Softwarerchitektur erhöht sich. Die Entwicklungszeiten werden verlängert und die Wartbarkeit der Architektur wird schwieriger.

Die integrierte Business-Logik hat weitere negative Nachwirkungen auf die Systemarchitektur, denn ein Re-Engineering der Architektur nur durch komplette Analyse des Gesamtsystems in vielen Prozessschritten ist möglich. Ein Re-Engineering der Systemarchitekur ist im angegebenen Zeitrahmen dieser Arbeit nicht machbar. Im nächsten Kapitel wird ein vollständiger Entwurf vorgestellt, dabei werden auch die möglichen Prozess-Schritte in Zukunft definiert. Eine Beispielanwendung, die komplex zu realisieren ist, wäre z. B. ein Callagent, bei dem die Call-Center Mitarbeiter die Möglichkeit haben sollen, telefonische Bestellungen anzunehmen und zu erzeugen. Die telefonischen Bestellungen sollen ähnlich wie die normalen Shopbestellungen abgewickelt werden. Die neue Anwendung soll dieselben Funktionalitäten nutzen, die im Shop-System implementiert sind. Speziell für E-Commerce Systeme werden sehr oft solche Anforderungen nachträglich eingeführt und die gleichen Funktionalitäten werden an mehreren Stellen mit unterschiedlichen Implementierungen integriert.

Ein weiteres Problem entsteht auch bei der Implementierung der Schnittstellen. Hier müssen die gleichen Schnittstellen vom Shop-System im CallAgent-System noch mal implementiert werden, damit die Bestellungen ohne zusätzlichen Aufwand im Fullfilment-System und im ERP-System abgearbeitet werden können. Aktuell versuchen die meisten E-Commerce-Unternehmen, ihre Produkte in Apps für mobile Geräte anzubieten, z. B. für iPhone, BlackBerry oder für Android Apps. Um diese Apps zu realisieren, soll für jede App die Shop-Logik dupliziert werden.

Diese Anforderung ist bei der aktuellen Systemarchitektur des Beispielunternehmens sehr schwierig und aufwändig zu realisieren, weil die logischen Funktionalitäten nicht modular oder als Extra-Komponente verfügbar sind. Die einzelnen Applikationen, bzw. Dienste, die mehrfach verwendet werden sollen, müssen mehrfach dupliziert werden und später mehrfach angepasst, bzw. aktualisiert werden.

4.3.2 Funktionale Anfoderungen

Das Ziel dieser Arbeit ist es, das Konzept für eine neue Softwarerchitektur zu entwickeln, welches beschreibt, wie die Realisierung einer SOA möglich ist. um die Integrationsproblematik und die darauf aufbauende Migrationsproblematik im Beispiel-Unternehmen zu erleichtern.

Dabei sollen die neuen Komponenten einfacher integrierbar sein und bestehende Komponenten sollen flexibler und wiederverwendbar sein. Dadurch wird die Wiederverwendung von Komponenten vereinfacht und die Entwiklungszeiten werden verkürzt. Die Wartbarkeit der Architektur wird durch die strukturierte Softwarearchitektur einfacher und übersichtlicher. Dadurch entsteht auch eine Reduktion der Entwicklungskosten.

Im nächsten Abschnitt werden die funktionalen Anforderungen dieser Arbeit in Form von Anwendungsfällen und Usecases verdeutlicht.

Anwendungsfälle aus Nutzersicht

Die folgende Abbildung 4.9 zeigt die Anwendungsfälle aus Sicht eines Nutzers, der das zu erstellende System beherrschen sollte.

Usecases

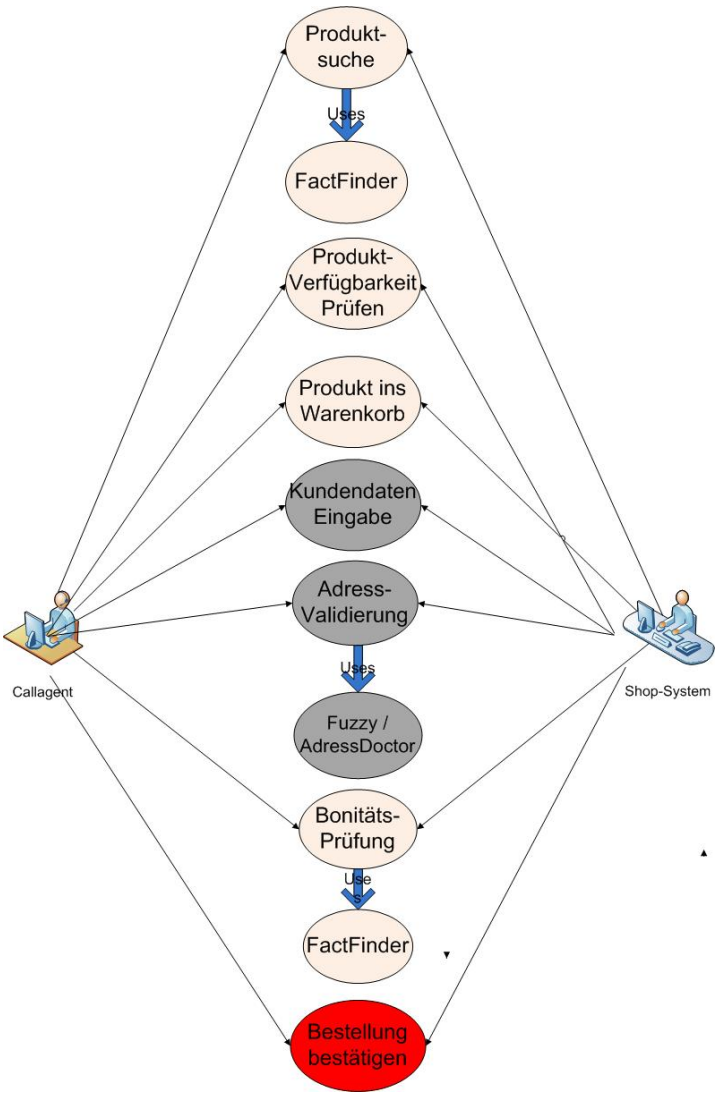

Abbildung 4.9: Anwendungsfälle aus Nutzersicht

Aktor	Beschreibung
Webshop User	Der User im Onlineshop, der bestellen soll. Tauschen von außen Informationen mit dem System aus
CallAgent User	Der Callagent User, der telefonische Bestellungen ins Callagent-System eingeben soll.
Webshop	Das Onlineshop. Für die Online-Annahme der Bestellung zuständig.
CallAgent	Das Callagent. Für die telefonische Annahme der Bestellung zuständig.
Mobile App	Mobile Applikationen, die Bestellungen aus mobilen Geräten annehmen können.

Use-Case 1

Anwendungsfall	**Produktsuche**
Aufgabe	Der Kunde gibt ein Suchwort ein, um passende Produkte zu finden
Beteiligte Aktore	Webshop User
Auslöser	Webshop User
Vorbedingung	keine
Nachbedingung	Die passenden Produkte werden gesucht und dargestellt.

Use-Case 2

Anwendungsfall	**Produkt ins Warenkorb**
Aufgabe	Der Kunde fügt ein Produkt ins Warenkorb
Beteiligte Aktore	Webshop User
Auslöser	Webshop User
Vorbedingung	Produkt ist verfügbar
Nachbedingung	Das Produkt wird dem Warenkorb hinzugefügt.

Use-Case 3

Anwendungsfall	**Adresse validieren**
Aufgabe	Der Kunde gibt seine Adresse im Bestellprozess ein
Beteiligte Aktore	Webshop User
Auslöser	Webshop User
Vorbedingung	Produkte im Warenkorb und der Kunde möchte bestellen.
Nachbedingung	Die Addresse wird beim richtigen Validierungsdienst validiert und eine Weiterleitung zur Zahlungsseite wird durchgeführt.

Use-Case 4

Anwendungsfall	**Bonität Prüfen**
Aufgabe	Die Kundendaten bei CreditReform prüfen lassen
Beteiligte Aktore	Webshop User
Auslöser	Webshop User
Vorbedingung	Produkte im Warenkorb und der Kunde hat eine richtige Adresse eingegeben.
Nachbedingung	Die ausgewählte Zahlungsart wird angenommen und die Bestellbestätigungsseite wird angezeigt.

Use-Case 5

Anwendungsfall	Bestellung Abschicken
Aufgabe	Der Webshop User bestätigt seine Daten und schickt die Bestellung ab.
Beteiligte Aktore	Webshop User
Auslöser	Webshop User
Vorbedingung	Gültige Kundendaten und Zahlungart
Nachbedingung	Die Bestellung wird angenommen und an das Fullfilmentsystem zwecks Auslieferung weitergegeben

Use-Case 6

Anwendungsfall	Webshop Bestellung abarbeiten
Aufgabe	Der Webshop User bestätigt seine Daten und schickt die Bestellung ab.
Beteiligte Aktore	Webshop
Auslöser	Webshop User
Vorbedingung	Bestellung angelegt
Nachbedingung	Die Bestellung wird an den BestellungsManager exportiert zwecks Weitergabe an das Fullfillmentsystem zwecks Auslieferung

Die funktionalen Anforderungen dieser Arbeit bilden die langfristigen Ziele der Unternehmen ab. Dabei ist das Hauptziel der Unternehmen, eine neue Softwarerchitektur zu schaffen, die die zukünftige Integration neuer Anwendungen mit möglichst wenig Aufwand, erleichtert. Die neu zu entwickelnde Softwarerchitektur soll auch die zukünftigen Entwicklungen, bzw. Anpassungen der Systeme beschleunigen. Im folgenden Abschnitt wird versucht, die Shopfunktionalitäten zu modularisieren und als logische Komponenten darzustellen. Danach findet eine Entscheidung statt, welche logische Komponenten im Shop-System bleiben und welche davon extra ausgelagert werden.

Shoplogik und Komponentenlogik

In der aktuellen Architektur ist die Anwendungslogik im Shopsystem implementiert (Abbildung 4.10). Eine Trennung der Anwendungslogik in logische Komponenten ist in der neuen Architektur vorgesehen, damit eine mehrfache Wiederverwendung der einzelnen Komponenten möglich wäre. Dabei werden Teile der Logik vom Shop-System getrennt und modularisiert (Re-Engineering). Dadurch werden die im Abschnitt 4.3.2 genannten funktionalen Anforderungen

erzielt und eine Basisarchitektur mit Komponenten wird dann geschaffen. Deshalb ist es notwendig, genauer zu ermitteln, welche logische Komponenten, bzw. Anwendungen von Shop-System getrennt werden können und welche nicht. Bei der Entscheidung, welche Komponenten vom Shop-System getrennt werden sollen, werden die einzelne Komponenten nach dem Sicherheitskriterum sortiert. Die sicherheitkritischen Komponenten, z. B.: Zahlungsgmodule, bleiben im Shop-System.

Die neuen Komponenten werden als Dienste angeboten, damit die beteiligten Systeme einen Zugriff auf diese Komponenten ermöglichen.

Wie in der Abbildung 4.10 zu sehen ist, sind folgende Funktionalitäten im Shopsystem implementiert:

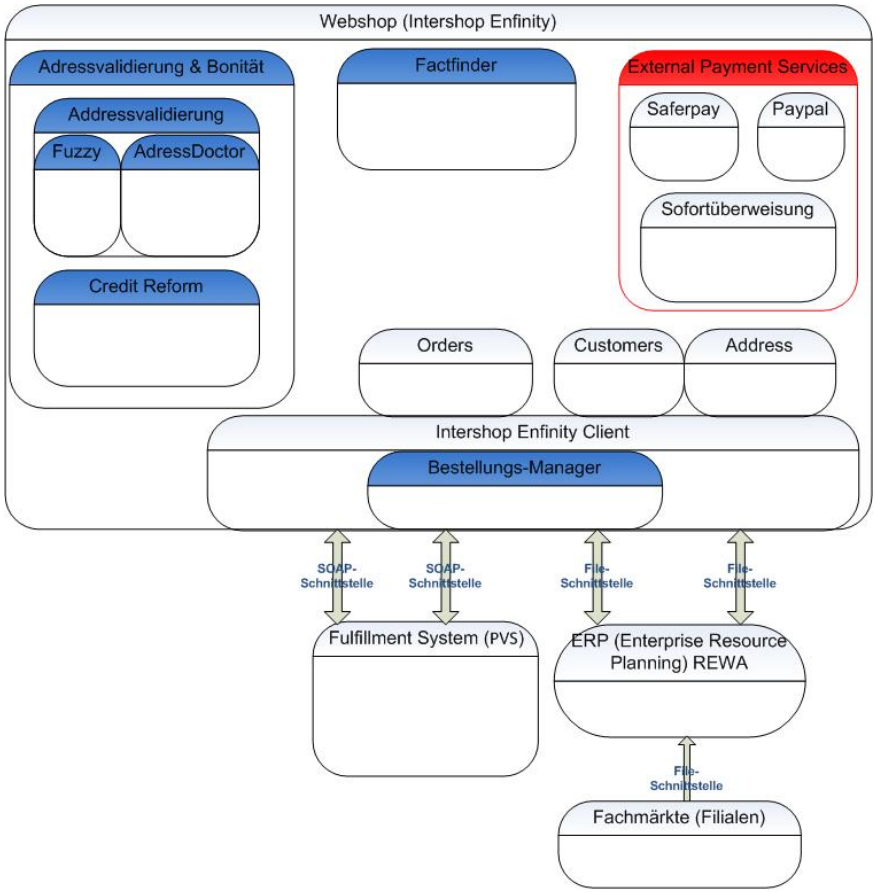

Abbildung 4.10: Shop-System Komponente

Wie in der Abbildung 4.10 zu sehen ist sind folgende Funktionalitäten im Shopsystem implementiert:

- Produktsuche
- Adressvalidierung
- Externe Zahlungssysteme
- Bonitätsprüfung
- Zusammenspiel Adressvalidierung und Bonitätsprüfung

Folgende Funktionalitäten werden umgelagert und als Extradienste angeboten, weil sie mehrfach verwendet werden können:

Produktsuche (FactFinder) Durch einen intelligenten Suchalgorithmus sollen die Produkte im Shop-System und im Callagent gefunden werden. Die Implementierung des Suchalgorithmus befindet sich im Shop-System. Zukünftig wird die Suche als Komponente entwickelt und ausgelagert werden, damit die Suche keine negative Wirkung auf die Performance des Shop-Systems entfaltet und um doppelte Entwicklungen zu vermeiden. Die Benutzer von CallAgent benötigen dieselbe Produktsuche als Feature im System, damit eine telefonische Beratung möglich ist.

Adressenvalidierung (AdressDoctor und Fuzzy) Adresseingaben im CallAgent sollen auch auf Richtigkeit geprüft und gegebenfalls korrigiert werden, damit die Sendungen nicht an falsche Adressen verschickt werden. Gerade bei den Validierungssystemen steckt im Shop-System eine Entscheidungslogik. Der Aufruf der Prüfdienste ist ebenfalls in der Shop-Applikation fest integriert. Die Entscheidung, welcher Dienst verwendet wird, ist abhängig von dem Zweck der Abfrage (Umkreissuche und/oder Rechnungsadressenprüfung) und von der Adresse. Deswegen wäre es sinnvoll, die Adressvalidierungssysteme zu integrieren.

BestellungsManager Für die neuen Vertriebswege werden immer neue Bestellungs schnittstellen zwischen den beteiligten Systemen benötigt. Um diesen Aufwand zu verringern, wird der BestellungsManager ausgelagert. Dabei übernimmt der Bestellungs-Manager die Kommunikation mit dem ERP- System und mit dem Fullfilment-System. Die neuen neuen Schnittstellen werden nur bei den neuen Vetriebswegen implementiert.

Bonitätsprüfung (CreditReform) Wenn ein Shop-System dem Kunden erlaubt, per Rechnung zu zahlen, dann ist diese Entscheidung eine Kreditentscheidung und erfordert eine Bonitätsprüfung des Kunden. Das E-Commerce-Unternehmen verwendet Creditreform, um die Bonität des Kunden zu ermitteln. Diese Prüflogik und der Aufruf des Bonitätsprüfdienstes sind ebenfalls in Shoplogik fest verdrahtet. In der CallAgent soll diese Prüfung

durchgeführt werden, besonders, wenn der Kunde auf Rechnung oder mittels Bankeinzug bezahlen will. Für die Bonitätsprüfung wird eine richtige Adresse vorausgesetzt. Diese Prüflogik könnte ebenfalls als externer Dienst angeboten werden, um eine Mehrfachverwendung zu ermöglichen.

Adressvalidierung und Bonitätprüfung Ein wichtige Anwendung mit integrierter Logik ist das Zusammenspiel von AdressDoctor, Fuzzy, Creditreform und Umkreissuche. Für die Entscheidung, welcher Validierungsdienst verwendet wird, wird das Herkunftsland des Users geprüft. Kunden aus Deutschland werden durch Fuzzy geprüft. Die Daten ausländischer Kunden werden durch AdressDoctor validiert. Für Kunden aus Deutschland werden für die Umkreissuche die Geodaten benötigt. Für solche Kunden wird AdressDoctor verwendet, damit bei der Ermittlung der Geodaten die Adresse validiert wird. Abhängig von dem Ergebnis der Adressenprüfung, wird entschieden, ob eine Bonitätsprüfung bei Creditreform durchgeführt werden soll oder nicht. Für Kunden, die eine möglicherweise falsche Adressen haben, wird keine Bonität geprüft, um zusätzliche Kosten zu sparen. Diese Entscheidungslogik, welcher Dienst verwendet werden soll, kann ebenfalls als externe Komponente ausgelagert werden, wenn die gleiche Logik in weitere Vertriebswege implementiert werden soll.

Im Abschnitt 4.3.2 wurden die einzelnen Funktionalitäten des Shop-Systems vorgestellt, die als Komponente dargestellt werden können.

Um den Zustand zu erreichen, die logische Funktionalität aus dem Shop-System herauszunehmen und als Komponente zu extrahieren, werden mehrere Schritte vorausgesetzt. Dabei wird ein Re-Engineering des Gesamtsystems durchgeführt.

Im ersten Schritt wird das System analysiert. Einzelne Funktionalitäten werden aufgelistet und die Abhängigkeiten werden dann definiert. Nichtkritische Funktionalitäten werden dann neu konzipiert und als Extra-Komponente extrahiert. Funktionalitäten, die voneinander abhängig sind, zum Beispiel das Zusammenspiel zwischen Fuzzy und AdressDoctor, werden zusammen in einer Komponente angeboten. Der meiste Aufwand bei dieser Aufgabe besteht zum großen Teil bei der Analyse des Altsystems (Kapitel 3.1.1). Aus den vergleichbaren Arbeiten wird der dritte Ansatz bevorzugt, dabei werden nur Teile der Legacy-Systeme neu zu entwickeln sein .

4.3.3 Nichtfunktionale Anforderungen

Nicht-funktionale Anforderungen gehören auch in den Service-Contract und haben Einfluss auf das Design und die Implementation. Betriebliche Aspekte, wie Monitoring und Lifecycle-Control dürfen nicht vergessen werden.

Bei der Entwicklung von SOA Architekturen sind folgende nicht-funktionale Anforderungen zu beachten:

Zuverlässigkeit und Reduktion der Komplexität Ein komplexes System ist meistens ein Legacy System, das historisch gewachsen ist. Das Beispielunternehmen besteht aus einer Architektur mit fest gekoppelten Komponenten. Sobald weitere Komponenten an die Architektur hinzugefügt werden, wird sich die Komplexität des Systems erhöhen. Ein Hauptziel der neuen Architektur ist aber, die Komplexität der Systeme zu vermeiden. Die Komplexität eines Systems soll möglichst niedrig bleiben bei geringeren Kosten und gestiegenen Qualitätsanforderungen.

Integrierbarkeit Das Hauptproblem in den meisten Legacy-Systemen besteht darin, neue Komponenten zu integrieren. Dadurch, dass die aktuelle Architektur aus fest gekoppelten Komponenten besteht, werden auch neue Komponenten, bzw. Anwendungen festgekoppelt und das Risiko, eine komplexe Systemarchitektur zu bekommen, wird höher. Daher gilt die Anforderung, einfache Integrierbarkeit in der neuen Architektur, entstehen zu lassen.

Wiederverwendbarkeit Die Komponenten in der aktuellen Softwarearchitektur sind jetzt festgekoppelt und eine Wiederverwendung ist nicht möglich. Zukünftig sollen die Komponeneten losgekoppelt werden, um die Wiederverwendbarkeit zu ermöglichen. Eine zusätzliche Anforderung an die neue Architektur wäre dann die Wiederverwendbarkeit.

Entwicklungszeiten und Kosten Festgekoppelte Komponenten haben weitere Nachteile, bzw. Folgeprobleme. Für die Integration neuer Anwendungen oder die Wiederverwendbarkeit bestehender Funktionalität werden längere Entwicklungszeiten benötigt und dadurch entstehen höhere Entwicklungskosten. Es lohnt sich meistens langfristig für das Unternehmen, einmalig in die Softwarearchitektur zu investieren, um später Zeit und Geld bei der Weiterentwicklung zu sparen.

Wartbarkeit Eine Folge der Anforderung der Komplexität ist die Wartbarkeit, denn nichtkomplexe Systeme sind meistens einfach wartbar. Im normalen Betrieb der Unternehmen müssen meistens Änderungen an der Software vorgenommen werden, sei es durch Programmfehler oder durch neue Anforderungen. Der Aufwand, ein bestehendes Softwaresystem je nach Bedarf anzupassen, ist unproblematischer, je abstrakter die Programmarchitektur angelegt wurde.

Skalierbarkeit Die Skalierbarkeit beschreibt das Verhalten von Systemen bei zunehmender Last, dabei bleibt die Performance des Systems stabil bei gleichzeitiger Erhöhung des Ressourcenbedarfs und wachsende Eingabemengen.

Robustheit Immer wieder ähnliche Konstruktionen wirken sich positiv auf die Qualität des Service aus.

Flexibilität Fähigkeit, schnell an neue Anforderungen angepasst zu werden, gilt im Kleinen (Service) und überträgt sich auf das Gesamte.

4.3.4 Zusammenfassung

Die Anforderungen an das neue System bestimmen die Randbedingungen für das neue Designkonzept.

In diesem Abschnitt wurde ein mögliches Beispiel-Szenario vorgestellt. Daraus wurden die allgemeinen funktionalen und nicht funktionalen Anforderungen formuliert. Das Verknüpfen von Services soll durch das ESB realisiert werden. Das System wurde modularisiert um die Entscheidung zu treffen, welche Komponente im Shop-System beibehalten werden können und welche davon als externe Komponente angeboten werden können. Das Designkonzept wird unter Betrachtung der funktionalen Anforderungen erstellt. Dazu gehört die einfache Erweiterbarkeit der Architektur. Zusätzlich werden die Zuverlässigkeit und Zuständigkeit der einzelnen Komponenten in der neuen Architektur bei dem neuen Designkonzept berücksichtigt.

Der Re-Engineeringsprozess des Altsystems wurde beschrieben und definiert, um die Entkoppelung der Funktionalitäten des Altsystems zu ermöglichen. Durch den Einsatz der neuen Architektur für lose gekoppelte Systeme werden diese Funktionalitäten für mehrere Vertriebswege verwendet.

Im nächsten Kapitel werden die technischen Voraussetzungen an Software und Hardware vorgestellt und genauer definiert. Dabei wird das ESB-Bus vorgestellt und eine Einführung gegeben, wie der ESB-BUS am besten in die neue Architektur integriert werden kann.

5 Design

Nach dem vorangehenden Kapitel über die Analyse des E-Commerce-Systems wird in diesem Kapitel ein Design erarbeitet, welches für den Entwurf eines neuen SOA-Modells notwendig ist. Die Hauptziele bestehen darin, ein System zu schaffen, das möglichst zuverlässig und einfach erweiterbar ist.

Es wird zunächst einen Überblick über die Systemarchitektur des E-Commerce-Unternehmens geben, in dem auch noch einmal die wesentlichen Komponenten des Systems kurz beschrieben werden. Dann werden die neuen Systemkomponenten ausführlich beschrieben. Die Anforderungen der Zielarchitektur werden im Abschnitt 5.1.1 erläutert.

Danach findet die Vorstellung des neuen Designkonzeptes statt. Dabei wird zuerst die Softwarearchitektur des JBoss ESB vorgestellt, um die technischen Konzepte und Grundlagen für den Entwurf zu erläutern. Dazu werden sowohl die clientseitigen Komponenten, als auch die serverseitigen Grundlagen für die Entwicklung von Services der neuen Module am Beispiel erläutert. Das Komponenten-Modell wird im Abschnitt 5.2.2 eingeführt.

Es wird gezeigt, wie sich Teile der Logik des Altsystems separieren lassen. Das Prinzip der Zuständigkeit wird durch Kohäsion und Auslagern von Datentransformation in Wrapperkomponenten realisiert. Die einfache Erweiterbarkeit wird durch Nutzung von Factories ermöglicht.

5.1 System-Architektur

In Abbildung 5.1 wird die aktuelle Systemarchitektur dargestellt, die verschiedenen Funktionalitäten werden hierbei als logische Module angezeigt.

Das Ziel dieser Arbeit besteht darin, die logischen Funktionalitäten aus dem Webshop System Intershop zu extrahieren, sie als separate Komponenten zu modellieren und die lose Koppelung der Komponenten auf Basis einer SOA zu ermöglichen. Durch diesen Ansatz wird eine

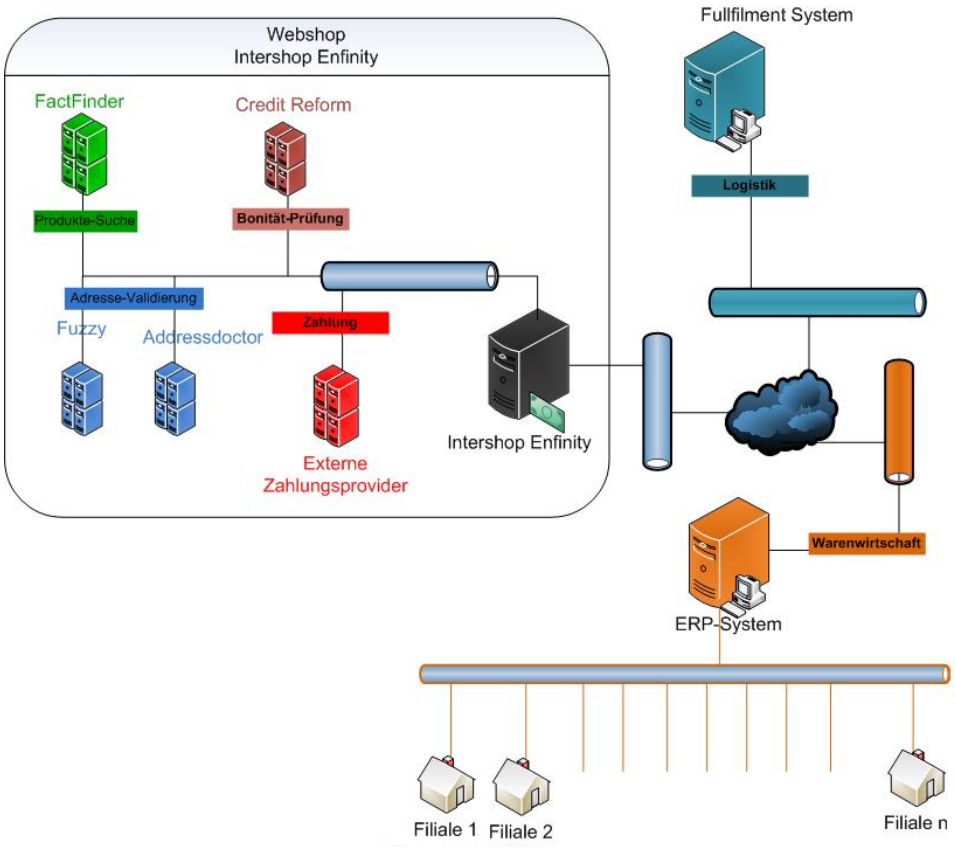

Abbildung 5.1: Systemarchitektur des Altsystems

einfache Integration der Module ermöglicht und auf neue Anforderungen, welche sich z. B. durch die Einführung eines neuen Vertriebs-Kanals ergeben, kann besser reagiert werden.

Folgende Funktionalitäten sind als logische Module in der Abbildung dargestellt:

- Bonitätsprüfung:

  Das E-Commerce-Unternehmen verwendet Creditreform, um die Bonität der Kunden zu ermitteln. Diese Prüflogik und der Aufruf des Bonitätsprüfdienstes lässt sich als logische Module abbilden (Abblildung 5.1).

- Produktsuche:

  Die Implementierung des Suchalgorithmus befindet sich im Shop-System (Factfinder). Durch die Auslagerung reduzieren sich die negativen Auswirkungen der Suche auf die

Performance des Shop-Systems, weil die Suche in den Shop-Systems am häufigsten aufgerufen wird und eine große Last auf die Performance verursacht.

- Adressenvalidierung:

  Derzeit werden zwei externe Dienste für die Adressenvalidierung verwendet (Fuzzy und AdressDoctor). So wird z. B. für deutschen Adressen Fuzzy verwendet und für Adressen aus Österreich wird AdressDoctor verwendet. Dazu müssen die Adressen des Web-Shops jeweils speziell in das Format des Prüfdienstes transformiert werden. Der Aufruf des Prüfdienstes ist in der Shop-Applikation fest integriert. Dieses Modul wird in Zukunft ausgelagert und als externe Komponente realisiert.

- Bestellungsmanager: Der Export der Bestellungen an das ERP-System oder an das Fullfillment-System, dazu der Import des Status aus dem Fullfillment-System ist eine logische Funktionalität im Shop-System und wird zukünftig auch vom Callagent oder von weiteren Systemen benötigt. Dieses Modul kann in Zukunft als externe Komponente ausgelagert werden, damit keine neue Entwicklungen in den weiteren Systemen benötigt werden.

Am Beispiel der Adressenvalidierung wird gezeigt, wie ein Re-Engineering basierend auf dem Prinzip der Zuständigkeit, die Wiederverwendung und Erweiterbarkeit bestehender fest integrierter Module als lose gekoppelte Komponente ermöglicht.

5.1.1 Zielarchitektur

Im vorherigen Kapitel wurden die funktionalen Anforderungen des Unternehmens definiert. In diesem Abschnitt wird ein Überblick über die Zielarchitektur präsentiert. Dabei befreit sich das Shop-System von der Aufgabe als Middleware, die vom ESB übernommen wird. Hierbei geschieht die Kommunikation innerhalb der Gesamtarchitektur und mit den externen Systemen auschließlich über das ESB, eine Ausnahme gibt es bei den Zahlungsdiensten, weil die Kommunikation mit dem Zahlungsprovider aus Sicherheitsgründe nur direkt geschehen darf. Die weiteren Funktionalitäten werden modular gemacht und mittels des ESB angesprochen. Bei der prototypischen Realisierung von SOA wird das Re-Engineering von Fuzzy im Bestellprozess durchgeführt.

Eine lose Koppelung von den zu integrierenden Systemen wird vom ESB realisiert. Dafür muss zuerst das ESB eingeführt werden. Danach wird eine standardisierte Schnittstelle zwischen dem ESB und dem Shop-System entwickelt, um die ausgelagerten Funktionen aufzurufen Abbildung 5.2 welche Komponenten ausgelagert und als extra Komponente Angeboten werden können. Diese Komponenten werden direkt mit dem ESB verbunden und stehen damit auch neuer Anwendung, wie z. B. dem Callagent oder dem mobilen Clients zur Verfügung.

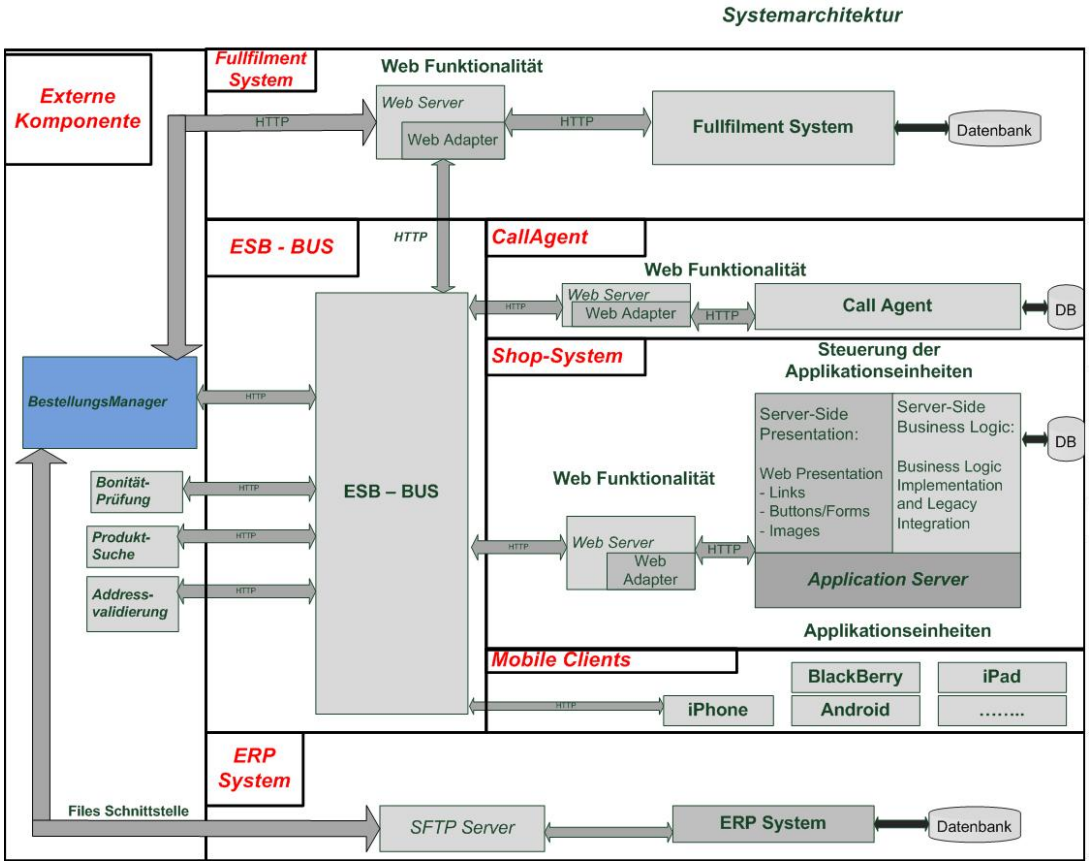

Abbildung 5.2: Architektur des Zielsystem

Die Abbildung zeigt, dass das Shop-System keine direkte Verbindung mehr mit dem Fullfillment- , ERP-System und zu den ausgelagerte Komponenten besitzt. Die ausgelagerten Komponenten sind heute feste Module, die extrahiert und losgekoppelt werden können.

Weitere Funktionalitäten können auch ausgelagert werden. Dazu muss eine Gesamtanalyse des Shop-Systems und des Callagent-Systems durchgeführt werden. Die gemeinsamen Funktionalitäten sollen ermittelt werden und können dann extrahiert und ausgelagert werden. Parallel zum Shop-System sind zwei neue Systeme (Callagent und Mobile clients) als Beispiel entstanden. Die zwei neuen Systeme sind, genauso wie das Shop-System, mit dem ESB verbunden und greifen auf dieselben Dienste zu. Durch die Auslagerung der Logik in Extrakomponenten können das Callagent und die Mobilen Clients darauf zugreifen, ohne die Gesamtfunktionalität doppelt zu implementieren.

Das Fullfillment- und ERP-System kommunizieren miteinander über das ESB.

Die Hauptaufgabe des ESB besteht darin, die Konnektivität zwischen den Systemen herzustellen und die Services zu verwalten. Es gilt als technisches Rückgrat einer SOA-Landschaft. Nach der Einführung des ESB wird die Integration der Komponenten erleichtert. Abbildung 5.2 gibt einen Überblick über das Ziel der Architektur.

Bei dem Re-Engineerungsprozess einer bestimmten Funktionalität wird im ersten Schritt die Logik dieser Funktion aus dem Shop-System Intershop ausgelöst. Im zweiten Schritt wird die neue Komponente realisiert, dann wird die Logik aus dem Shop-System in die neue Komponente integriert. Zuletzt soll ein Aufruf dieser Funktionalitäten durch einen ESB ermöglicht werden und der Aufruf ins Shop-System eingebaut werden.

Wie das Auslösen der Adressenvalidierungs-Logik aus dem Shop-System durchgeführt werden soll und wie diese Logik in den AdressValidator integriert werden kann, dazu, welcher Ansatz am besten verwendet werden kann, wird im Abschnitt Integrationsaufgabe (5.1.2) genauer definiert.

5.1.2 Integrationsaufgabe

Das Anwendungsbeispiel basiert auf das vorgestellte Use-Case vom Kapitel 4.3.2. Dabei wird eine neue Komponente erstellt, die Validierungslogik aus dem Shop ausgelagert und anschließend wird die Integrationsaufgabe durchgeführt. Zunächst wird entschieden, welcher Integrationsansatz am besten geeignet ist. Es existieren hauptsächlich drei unterschiedliche Integrationsansätze, um eine Integrationsaufgabe durchzuführen.

Bei dem ersten Ansatz wird die existierende Applikations-Infrastruktur beibehalten und nur die Schnittstellen werden als Webservices entwickelt (Kapitel 3.1.1).

Beim zweiten Anzatz wird das Legacy-System neuentwickelt und in Services integriert (Kapitel 3.1.1).

Der dritte Ansatz ist eine Mischung von den zwei ersten Ansätzen, dabei werden nur Teilkomponenten des Legacy-Systems neu entwickelt (Kapitel 3.1.1).

Bei dem Anwendungsbeispiel geht es hauptsächlich darum, die Adressenvalidierungs-Logik auszulagern. Wenn der Re-Engineering-Ansatz verwendet wird, wird die komplette Logik neu programmiert. Dadurch entsteht der Nachteil, dass die Entwicklungszeit länger dauert und die Entwicklungskosten deutlich höher werden.

Wenn der Wrapping-Ansatz verwendet wird, wird die Entwicklungszeit kürzer und die Entwicklungskosten werden geringer. Der Nachteil, dass die Zukunftige Wartungen komplizierter werden, kann bei diesem Beispiel-Szenario nicht entstehen, denn die Logik der Adressenvalidierung, die umgelagert werden soll, ist sehr einfach und verständlich für den Entwickler.

Beim dritten Ansatz werden nur Teilkomponenten des Adressvalidierungssytems neu entwickelt (Kapitel 3.1.1). Dieser Ansatz kann für die komplette Einführung eines SOA in ein E-Commerce-System dienen, denn dabei wird das gesamte System in Komponenten zerlegt und dieser Ansatz entscheidet, welche davon ein Re-Engineering benötigen und bei welchen ein Wrapping-Ansatz reichen würde.

Beim Beispiel-Szenario wird die neue Komponente entwickelt. Für die Integration dieser Komponente wird der Wrapping-Ansatz bevorzugt, denn dieser Ansatz ist unkompliziert und besser geeignet für die Integration. Bei dieser Integration wird die Logik der Adressvalidierungskomponente umgelagert und für SOA zur Verfügung gestellt.

Die Funktionalitäten dieser Komponente werden nur noch über das von SOA verwendete JBoss ESB aufgerufen. Ein Migrationsprozess wird auch nicht benötigt, da bei der Adressenvalidierung keine Daten in der Datenbank gespeichert werden.

Bei der Einführung der SOA-Architektur werden weitere Aspekte mit einbezogen. Dabei stellt das Enterprise Service Bus (ESB) die Kernkomponente oder das technische Hilfsmittel einer SOA dar.

Die neuen Komponenten werden hinter dem ESB festgekoppelt und können gleichzeitig von mehreren und unterschiedlichen Anwendungen verwendet werden.

Der Aufruf der Komponente erfolgt zukünftig über das ESB. Im folgenden Abschnitt werden die Funktionalitäten des JBoss ESBs kurz vorgestellt. In dieser Arbeit wird für die Realisierung der ESB von JBoss gewählt.

5.2 Entwurf

In diesem Abschnitt wird zuerst die Softwarearchitektur des JBoss ESB Bus vorgestellt, um die technischen Konzepte und Grundlagen für den Entwurf zu erläutern. Dabei werden sowohl die clientseitigen Komponenten, die für die Verwendung der lose gekoppelten Module des neu entwickelten Ziel-Systems relevant sind, als auch die serverseitigen Grundlagen für die Entwicklung von Services der neuen Module am Beispiel erläutert. Abschnitt 5.2.2 führt das Komponenten-Modell in diese Arbeit ein, Abschnitt 5.2.3 weist anhand von kommentierten Klassendiagrammen die logische Entkoppelung der Funktionalität des Altsystems aus.

Standardaktionen werden vorgestellt und neue Aktionen werden designed. Im Anschluss werden die wichtigen Komponenten des Shop-Systems in Form von Klassendiagrammen dargestellt. Die Abhängigkeit zwischen den Komponenten und den verwendeten Methoden werden ebenso vorgestellt.

5.2.1 JBoss ESB

Für den Betrieb der Interaktion zwischen den Services und mit den bestehenden Systemen bedarf es einer Kommunikations-Infrastruktur. Der JBoss ESB stellt diese Infrastruktur zur Verfügung, dabei werden drei Hauptkomponenten unterschieden:

Service Registry ist eine JAXR Registry-Implementation.

JMS Gateway Listener Der „Gateway Listener“ ist eine der Schlüssel Architektur-Komponenten von JBossESB. Dieser Typ Listener ist die Schnittstelle zu dem ESB aus Endpoints außerhalb der ESB-Domain.

ESB Aware and Unaware Messages In dem JBossESB-Konzept ist der Typ „Message“ genau definiert, nämlich in xml/message.xsd. Mit diesem Konzept wird die Übertragung von decorated messages zwischen den Komponenten des ESB ermöglicht.

Die Besonderheit bei der JBoss ESB liegt in der Form der Kommunikation und der Transportmöglichkeiten, dabei werden Messaging-Services verwendet, wie z.B. HTTP, E-Mail und JMS (JBossMQ, JBoss Messaging, IBM MQSeries und ActiveMQ). Weiter besitzt ein JBOSS ESB eine Service Registry, das Persisted Event Repository für die Verwaltung der ESB-Umgebung und einen Notification Service für die Registrierung von Events und um Teilnehmern zu melden.

Weitere ESB-Funktionalitäten werden auch von JBOSS ESB angeboten, z. B. die Unterstützung von Datentransformationen durch Verwendung von Smooks oder XSLT, ein Content-basiertes Routing unter Verwendung von JBoss Rules oder XPath.

Services im JBOSS ESB sind als Liste von mehreren Aktion-klassen definiert. Dabei wird die Aktionsliste in Form einer „Action Pipeline“ dargestellt (Abbildung 5.3).

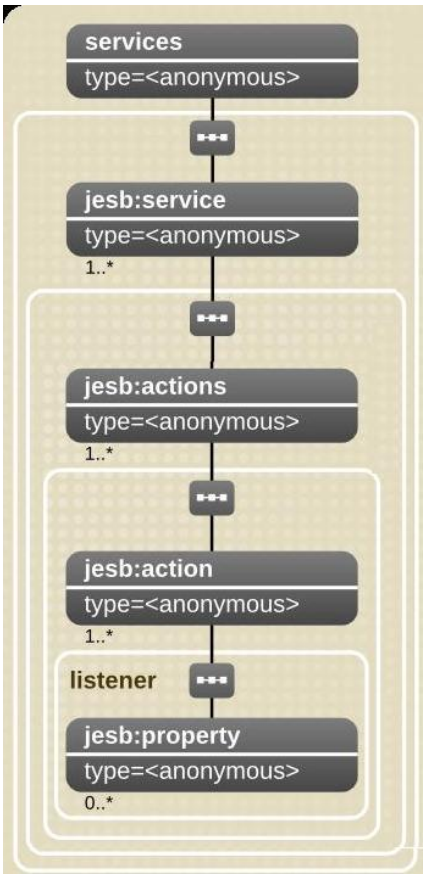

Abbildung 5.3: Services im JBoss

Ein Service kann gleichzeitig mehreren Aktionen durchführen. Die Servicedefinition erfolgt in der jboss-esb.xml Datei. Im Folgenden wird eine Beispieldatei vorgestellt:

```
<?xml version = "‘1.0"’ encoding = "‘UTF-8"’?>
<jbossesb xmlns="’http://anonsvn.labs.jboss.com/labs/jbossesb/
trunk/schemas/xml/jbossesb-1.0.1.xsd"’ invmScope="’GLOBAL"’>
<services>
<service category="’ServiceCategory"’ name="’ServiceName"’ >
<actions>
<action name="action1" class=
```

```
"org.jboss.soa.esb.actions.SystemPrintln">
 <property name="message" value="action 1 ausführen"/>
</action>

<action name="action2" class=
"org.jboss.soa.esb.actions.SystemPrintln">
 <property name="message" value="action 2 ausführen"/>
</action>

<action name="action3" class=
"org.jboss.soa.esb.actions.SystemPrintln">
 <property name="message" value="action 3 ausführen"/>
</action>

</actions>
</service>
</services>
</jbossesb>
```

Die vorgestellte XML-Datei definiert ein JBoss Service mit dem Namen „ServiceName“ aus der Kategorie „ServiceCategory“. Dieser Service besteht aus drei Aktionen, die jeweils ein System.out ausführen. Die dargestellten Aktionen können synchron und auch asynchron durchgeführt werden.

Bei dem Deployment des Service wird der Name und die Category dieser Services für die Registrierung in der Repository verwendet.

Die Client Anwendung kann den JBoss-Service auf folgende Weise finden:

```
ServiceInvoker invoker =
new ServiceInvoker("`ServiceCategory"', "`ServiceName"');
```

Der ServiceInvoker sucht den Service in der Service-Repository durch den Servicenamen und die ServiceCategory. Typischerweise wird nach der Auffindung des Service eine Nachricht in Form eines Message-Objektes angelegt und auf folgende Weise an den Service geschickt:

```
Message message = MessageFactory.getInstance().getMessage();
message.getBody().add("`Hier Message eingeben!"');
....
invoker.deliverAsync(message);
```

Die Nachricht kann auf zwei Weisen ausgeführt werden:

- Asynchron

  ```
  invoker.deliverAsync(message);
  ```

  Dabei wird dieselbe Nachricht asynchron von allen Service-Aktionen ausgeführt.

- Synchron

  ```
  Message responce;
  long timeout= 20001;
  responce = invoker.deliverSync(message, timeout);
  ```

  Beim synchronen Aufruf muss ein Timeout gesetzt werden. Die Nachricht wird synchron von allen Aktionen ausgeführt und bearbeitet und am Ende wieder als Responce zurückgegeben.

Die Kommunikation mit dem ESB läuft ausschließlich über Messages (Abbildung 5.4).

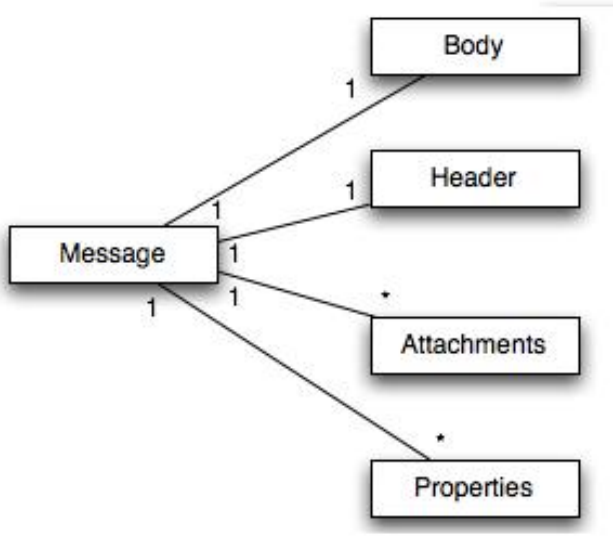

Abbildung 5.4: Messages Aufbau im JBoss

Diese Messages sind Implementierungen der Interface org.jboss.soa.esb.message.Message:

```
public interface Message
{
    public Header getHeader ();
```

```
    public Context getContext ();
    public Body getBody ();
    public Fault getFault ();
    public Attachment getAttachment ();
    public URI getType ();
    public Properties getProperties ();
    public Message copy () throws Exception;
}
```

Der Body-Teil des Message Objekts beinhaltet typischerweise die Nutzdaten dieser Nachricht und implementiert folgende Interface:

```
public interface Body
{
public static final String DEFAULT_LOCATION =
 "org.jboss.soa.esb.message.defaultEntry";
    public void add (String name, Object value);
    public Object get (String name);
    public byte[] getContents();
    public void add (Object value);
    public Object get ();
    public Object remove (String name);
    public void replace (Body b);
    public void merge (Body b);
  public String[] getNames ();
}
```

Der JBoss ESB unterstützt standrdmäßig folgende Interfaces als Erweiterungen der Body Interface:

org.jboss.soa.esb.message.body.content.TextBody Der Inhalt des Body Objects ist ein Text und kann durch die Methoden getText und setText gesetzt, bzw. ermittelt werden.

org.jboss.soa.esb.message.body.content.ObjectBody Der Inhalt des Body Objektes ist ein serialisiertes Object und kann durch die Methoden getObject und setObject gesetzt bzw. ermittelt werden.

org.jboss.soa.esb.message.body.content.MapBody Der Inhalt des Body Objectes ist ein Map<String, Serialized) mit einem String als Key und einem serialisierten Objekt als Value und kann durch die Methoden setMap und getObject gesetzt bzw. ermittelt werden.

Die Standard-JBoss-Aktionen, die mehrere Parameter erwarten und zurückliefern, verwenden standardmäßig das MapBody, damit mehrere Parameter eingegeben und zurückgeliefert werden können.

Benutzerdefinierte Aktionen können auch angelegt werden, dabei sollen diese Aktionen die Klasse AbstractActionPipelineProcessor überschreiben, z. B. :

```
public class TestAction extends AbstractActionPipelineProcessor {
  public void initialise() throws ActionLifecycleException {
     // Initialize resources...
     }
  public Message process(final Message msg) throws
  ActionProcessingException {
   //Nachricht Inhalt ermitteln
   MapBody mb=(MapBody)msg.getBody().get();
        // Process messages in a stateless fashion...
     }
  public void destroy() throws ActionLifecycleException {
         // Cleanup resources...
     }
}
```

Defaultmäßig wird die Methode process ausgeführt, wenn keine andere Methode definiert ist. Die Geschäftslogik kann innerhalb dieser Aktion implementiert werden. Als Rückgabewert der Methode process wird ein Message Objekt zurückgegeben, weil die Kommunikation innerhalb der JBoss ESB ausschließlich über Messages durchgeführt wird.

Standard JBoss Funktionalitäten werden auch in Form von Aktionen angeboten. Im Rahmen dieser Arbeit werden folgende Funktionalitäten benötigt:

SOAPProcessor Um Webservices zu publizieren wird die Aktion SOAPProcessor verwendet, dabei erwartet diese Aktion als Parameter den Webservice Namen:

```
<action name="JBossWSAdapter"
class="org.jboss.soa.esb.actions.soap.SOAPProcessor">
<property name="jbossws-endpoint" value="WebServiceName"/>
</action>
```

Der Parameter jbossws-endpoint ist ein Pflichtparameter beim Aufruf dieser Aktion, denn der Wert dieser Parameter beschreibt den Webservice-Namen, der publiziert werden soll. Im Folgenden wird ein Beispiel vorgestellt, wie ein Service definiert werden kann:

```
@WebService(name = "WebServiceName",
targetNamespace="http://ws_producer/adrprocessorWS")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class WebServiceName {

@WebMethod
public String processAddress
(@WebParam(name="eingabeParameter") String eingabeParameter) {

  Message esbMessage = SOAPProcessor.getMessage();
  if(esbMessage != null) {
      System.out.println(esbMessage.getBody().get());
  }
  System.out.println("WS Parameter=" + eingabeParameter);
  return "... returned:!! - " + eingabeParameter;
    }
}
```

Beim Aufruf dieser Aktion wird der Webservice mit dem Namen „WebserviceName“ gesucht, publiziert und in einer WSDL-Datei beschrieben. Dabei wird der Service als Standard beschriebener SOAP Webservice publiziert. Folgende Abbildung beschreibt, wie ein SOAP Webservice durch die SOAPProcessor Action publiziert wird:

In der Abbildung wird auf dem JBoss ESB die SOAPProcessor Aktion ausgeführt, um den Webservice auf dem JBoss WS zu publizieren. Der publizierte Web Service kann direkt von SOAP Clients ohne Hilfe vom JBoss ESB angesprochen werden.

SOAPClient Um Webservices-Clients zu entwickeln wird die SOAPClient Aktion verwendet. Dabei erwartet diese Aktion den Pfad der WSDL Datei und den Namen der Operation, die durchgeführt werden soll. Aus der WSDL Datei werden die Java Klassen erzeugt und verwendet. Optional können für diese Aktion folgende Eigenschaften definiert werden, falls der Webserviceanbieter eine Authentifizierung voraussetzt:

- username
- password

```
<action name="soap-client-action"
class="org.jboss.soa.esb.actions.soap.wise.SOAPClient">
  <property name="wsdl" value="http://host/WebserviceName?wsdl"/>
    <property name="operationName" value="processAddress"/>
</action>
```

Processor.JPG Processor.JPG

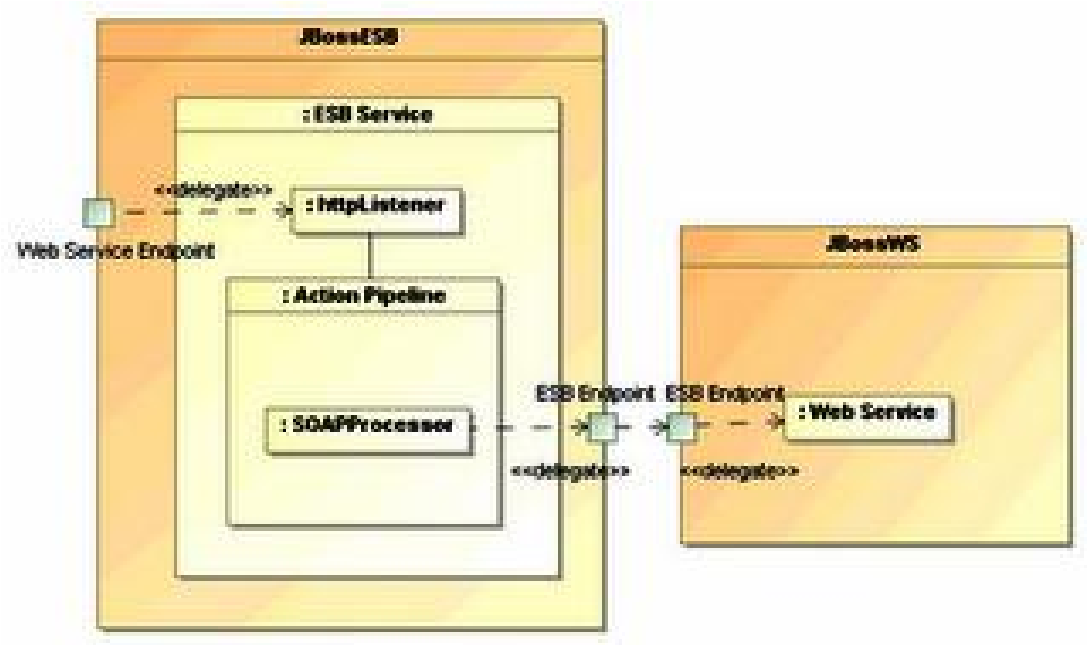

Abbildung 5.5: SOAP Processor Aktion

Beim Aufruf dieser Aktion wird ein SOAP Envelope mit allen Parameter aus dem Body-Teil der Message Objekt erstellt. Im Folgenden wird ein Beispiel mit drei Aktionen vorgestellt, welche die Webservice Parameter vorbereiten, den Webservice aufrufen und die Antwort auslesen:

In der jboss-esb.xml wird der Service „TestSOAPClient“ definiert, der drei Aktionen durchführen soll:

```
<services>
<service category="’SOAPClients"’ name="’TestSOAPClient"’ >
<actions>´

<action name="RequestAction" class="org.RequestAction"></action>
<action name="soap-client-action"
class="org.jboss.soa.esb.actions.soap.wise.SOAPClient">
  <property name="wsdl" value="http://host/WebserviceName?wsdl"/>
    <property name="operationName" value="processAddress"/>
</action>
<action name="ResponceAction" class="org.ResponceAction"></action>
```

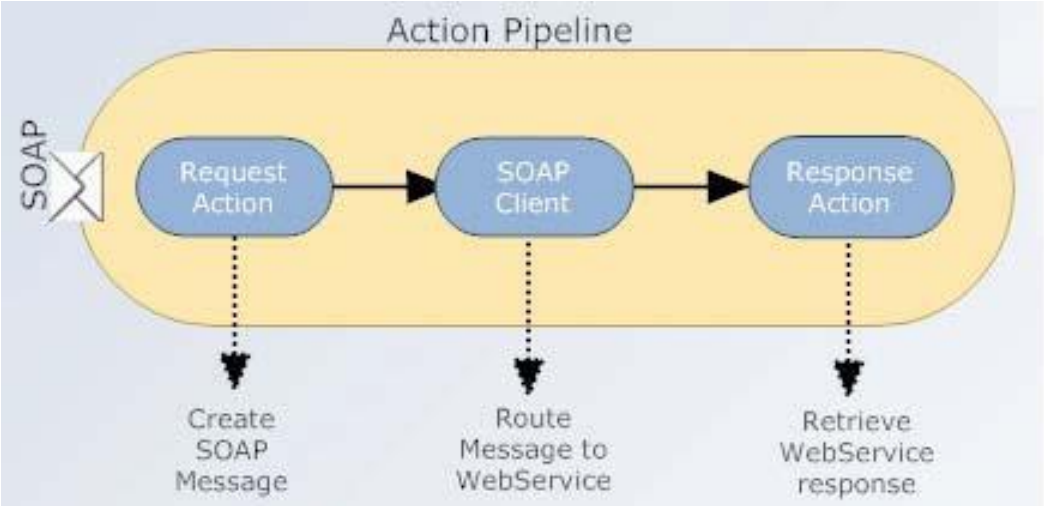

Abbildung 5.6: SOAP ClientAction

```
</actions>
</service>
</services>
```

Der Aufruf dieses Service wird synchron auf folgende Weise durchgeführt, damit alle drei Aktionen in der richtigen Reihenfolge ausgeführt werden und damit das Validierungsergebnis zurückgeliefert werden kann:

```
ServiceInvoker invoker =
new ServiceInvoker("'SOAPClients"', "'TestSOAPClient"');
Message message = MessageFactory.getInstance().getMessage();
message.getBody().get().put("'Street"','Brahmsallee"');
message.getBody().get().put("'StreetNo"',"'41"');
message.getBody().get().put("'City"',"'Hamburg"');
message.getBody().get().put("'Zip"','201442);
Message responce= invoker.deliverSync(message,5000l);
```

Im Folgenden werden die Klassen RequestAction und ResponceAction vorgestellt. Dabei hat die Klasse RequestAction die Aufgabe, die Parameter Street und StreetNo zusammenzufügen und in dem Message Objekt zu aktualisieren. Die Klasse ResponseAktion hat die gegen Aufgabe, beide Parameter zu trennen:

```
public class RequestAction extends AbstractActionPipelineProcessor {
  public Message process(Message msg) throws
  ActionProcessingException {
   //Nachricht Inhalt ermitteln
   MapBody mb=(MapBody)msg.getBody().get();
    // Parameter für den WebserviceAufruf vorbereiten
   String newStreet= mb.get("`Street"')+"' "'+ mb.get("`StreetNo"');
    mb.put("`Street"',newStreet);
    return msg;
     }
}

public class ResponceAction extends AbstractActionPipelineProcessor {
  public Message process(Message msg) throws
  ActionProcessingException {
   MapBody mb=(MapBody)msg.getBody().get();
      // Parameter aus dem Body Teil auslesen und bearbeiten
      String newStreetNo= (String)mb.get("`Street"')
      .subString(String.IndexOf("` "'));
      mb.put("`StreetNo"',newStreetNo);
      return msg;
     }
}
```

SystemPrintln Die SystemPrintln Aktion wird um das Debbuging während der Entwicklung benötigt. Die SystemPrintln wird durchgeführt wenn diese Aktion in der jboss-esb.xml definiert ist:

```
<action name="print" class=
"org.jboss.soa.esb.actions.SystemPrintln">
<property name="message"
   value="Dieser Textnachricht wird ausgegeben"/>
</action>
```

In diesen Abschnitt wurde eine kurze Einführung ins Jboss Esb gegeben. Dabei wurden die ESB-Messages vorgestellt, die für die Kommunikation zwischen dem ESB und den einzelnen Komponenten zuständig sind sowie das Modell für die Server und die Client Entwicklung vorgestellt. Im nächsten Abschnitt wird das Komponentenmodell dargestellt, gefolgt von den Klassendiagrammen der einzelnen Komponenten.

5.2.2 Komponenten-Modell

Bei der Analyse des Shop-Systems (Abschnitt 4.3.2) hat die Entscheidung stattgefunden, welche Funktionalitäten vom Shop-System getrennt werden können und als logische Komponenten angeboten werden können. Folgendes Komponentendiagramm (Abbildung 5.7) gibt einen Überblick über diese Komponente.

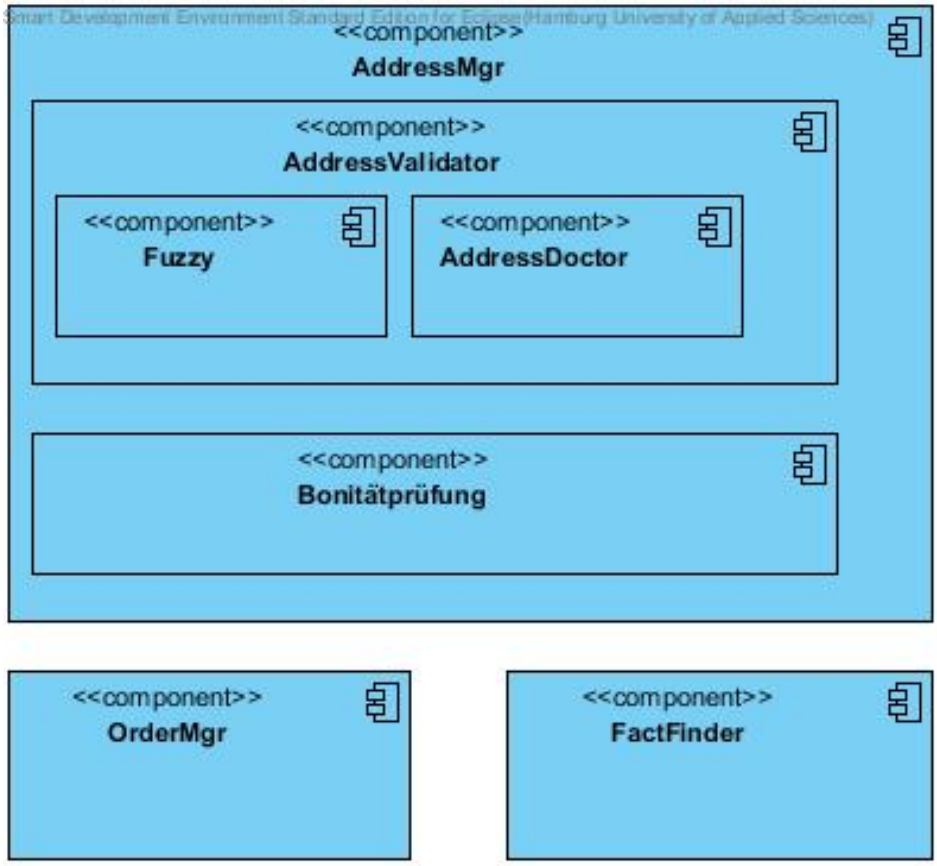

Abbildung 5.7: Komponenten Diagramm

Am Beispiel der AdressValidator Komponente wird im folgenden gezeigt, wie durch logische Entkoppelung der Funktionalität und durch Nutzung der ESB Funktionalitäten, eine lose Koppelung von Kernkomponenten eines Web-Shops erzielt werden kann. Dabei findet eine Modellierung des AdressValidator Service und Client statt.

5.2.3 Servicemodellierung

Der AdressValidator Service bietet die Möglichkeit, Adressen von unterschiedlichen Validierungsdiensten zu validieren. Somit kann der Service beispielsweise mit zusätzlichen Validierungsdiensten erweitert werden oder die bestehenden austauschen. Bei einem Service sind zwei Rollen zu berücksichtigen: der Servicenutzer und der Serviceanbieter. Der Service-Nutzer in dieser Fallstudie wird der JBoss ESB und der Serviceanbieter sind die externen Komponenten. Abbildung 5.8 zeigt das Klassenmodell der Serveranwendung, deren Klassen im Folgenden erläutert werden. Dabei wird auf die verwendeten Designmuster eingegangen.

Daran schließt sich die Realisierung der Klassen im JBoss ESB Framework an. Im Folgenden wird das Klassenmodell des AddressValidator Service vorgestellt. Dabei werden die Klassendiagramme client- sowie serverseitig definiert, bzw. dargestellt:

1. **AddressValidator Server** Aus technischer Sicht betrachtet muss aus dem bereits existierenden Validierungsdienst eine Schnittstelle als Webservice entwickelt werden. Insofern muss der Service auf sprachneutrale Weise für die Servicenutzer zur Verfügung stehen Hierzu bietet es sich an, eine WSDL zu definieren, die den Service beschreibt. Die Abbildung 5.8 stellt das Klassenmodell des Services dar. Dabei ist die Klasse AddressProcessorWS mit dem Stereotyp Webservice versehen. Server seitig befindet sich der Aufruf der Validierungsdienste und die Validierungslogik.

   Dabei besteht der AddressValidator-Server(Abbildung 5.8) aus folgende Klassen:

   Address: Diese Klasse beinhaltet das Modell für Addresse, das vom AddressModell der Validator abstahiert. Dabei wird das Interface serialized implementiert. Dieses Modell soll außerdem alle möglichen Eigenschaften enthalten, die eine Addresse haben können.

   Validator: Diese Klasse ist das Interface der Validierungsdieste, dabei werden die einheitlichen Methoden definiert.

   FuzzyValidator: Diese Klasse ruft den Validierungsdienst Fuzzy auf, dabei wird die Request-XML generiert und die Responce-XML interpretiert. Um die Zuständigkeit der Anwendung zu gewährleisten und die Erweiterbarkeit des AddressValidator zu ermöglichen wird als Eingabe das ObjektModell Address erwartet. Die Address-Transformation zu einer Fuzzy-Addresse finden in dieser Klasse statt.

   AddressDoctorValidator: Diese Klasse ruft den Validierungsdienst AddressDoctor auf. Um die Zuständigkeit der Anwendung zu gewährleisten und die Erweiterbarkeit des AddressValidator zu ermöglichen erwartet die AddressDoctorValidator-Klasse als Eingabe das ObjektModell Address. Die AddressTransformation zu ei-

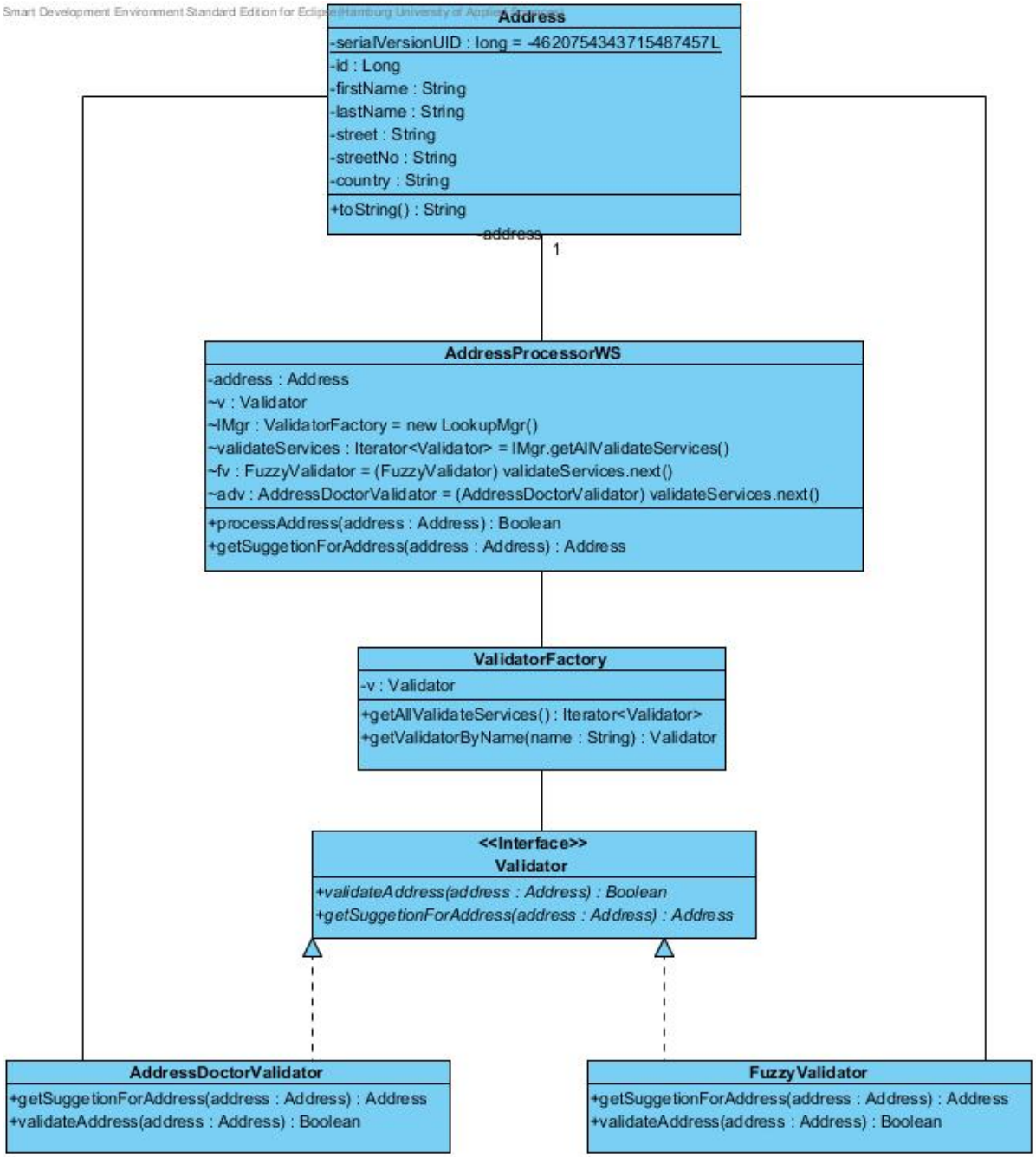

Abbildung 5.8: Server-Anwendung der AddressValidator

ner AddressDoctor-Addresse, das Generieren der SOAP-Request und das Auslesen der SOAP-Responce finden in dieser Klasse statt.

ValidatorFactory: Die ValidatorFactory ist ein Hilfsmittel zur Ermittlung und Erzeugung von Validatoren, unabhängig von einer konkreten Validator-Klasse. Sie wird verwendet, um die Instanziierung über den new()-Operator zu vermeiden, weil das Validator Objekt zu dem Zeitpunkt noch nicht bekannt ist. Diese Factory prüft und ermittelt, welche Validierungsdieste zur Verfügung stehen. Bei dem Entwurf dieser Muster wurde die design pattern factory methode verwendet, um die Erweiterbarkeit zu ermöglichen.

AddressProcessorWS: Diese Klasse beinhaltet die Webservice-Methoden, die angeboten werden können, zusätzlich die Validierungslogik, z. B. welcher Dienst verwendet werden kann, dazu den dynamischen Aufruf dieses Dienstes. Diese Methode wird als Wrapper der extrahierten Logik aus dem Altsystem verwendet.

Dabei wird die Logik übernommen und an das neue Design Pattern angepasst. Das neue Adress Modell wird auch in dieser Klasse verwendet, um die Erweiterbarkeit der Anwendung zu ermöglichen.

Für die Publizierung der Webservice wird nur die Webservice Klasse AddressValidatorWS benötigt und um die Publizierung durchzuführen wird die JBoss Aktion SOAPProcessor verwendet. Dabei erwartet diese Aktion als Parameter den Webservice-Namen und wird in der jboss-esb.xml definiert:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<jbossesb xmlns="http://anonsvn.labs.jboss.com/labs/jbossesb
/trunk/product/etc/schemas/xml/jbossesb-1.0.1.xsd"
        parameterReloadSecs="5">
<services>

<service category="ValidatorCategory" name="WSAddressService"
 description="WS Description">
<actions>

<action name="JBossWSAdapter"
class="org.jboss.soa.esb.actions.soap.SOAPProcessor">
    <property name="jbossws-endpoint" value="AdrValidatorWS"/>
</action>

</actions>
```

```
</service>
</services>
```

Diese Aktion setzt die Klasse AddressValidatorWS voraus, die den Webservice „AdrValidatorWS“ implementieren soll:

```
@WebService(name = "AdrValidatorWS",
targetNamespace="http://ws_producer/adrvalidatorWS")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class AddressValidatorWS {

@WebMethod
public Address processAddress
(@WebParam(name="address") Address address) {
//hier wird die validierungslogik implementiert z. B.
Address adr=Validator.validate(address);
    return adr;
    }
}
```

Aus dieser Klasse und jboss-esb.xml, welches die Webservice Aktion definiert und die benötigte Methoden beinhaltet, kann der Webservice publiziert werden. Die Auffindung und Publizierung des JBoss-Service wird auf felgende Weise durchgeführt:

```
ServiceInvoker invoker =
new ServiceInvoker("`ValidatorCategory"', "`WSAddressService"');
Message message = MessageFactory.getInstance().getMessage();
message.getBody().get().put("`jbossws-endpoint"',"'AdrValidatorWS"');
invoker.deliverAsync(message);
```

Der publizierte Web Service ist der zentrale Punkt für alle Validierungsabfragen und nimmt sämtliche SOAP-Anfragen via HTTP auf Port 80 entgegen und versendet auf dem gleichen Wege die Antworten an die Clients. Mit Hilfe der SOAPProcessor Action lässt sich mit Hilfe einer Build Datei (Apache ANT [APACHEANT]) ein AxisArchiv (aar) erzeugen. Die Archiv-Datei wird dann in eine WSDL konvertiert und veröffentlicht. Die WSDL-Datei wird auf der Client-Anwendung benötigt, um die Service-Eigenschaften und -Methoden zu definieren. Im nächsten Abschnitt findet eine Vorstellung dieser Client-Anwendung statt.

1. **AddressValidator Client**

   Nach der Vorstellung der serverseitigen Klassendiagramme der AddressValidator werden in diesem Abschnitt die clientseitigen Klassendiagramme vorgestellt.

   Der AddressValidator-Client(Abbildung 5.9) ist für die eigentliche Übertragung der Daten zum Web Service zuständig. Es besteht aus folgenden Klassen:

   MyRequestAction: Diese Klasse beinhaltet die JBoss Aktion für das Generieren der Requests. Dabei wird ein Address Objekt vorbereitet und als Message an die Server Anwendung gesendet.

   MyResponceAction: Diese Klasse beinhaltet die JBoss Aktion für das Auslesen der Responce Message. Dabei wird die Responce Message ausgelesen, interpretiert und daraus das Address Objekt extrahiert. Das Address Objekt wird nur dann ausgelesen, wenn ein AddressVorschlag zurückgeliefert ist.

   ProcessAddressRequest: Diese Klasse ist für das Generieren des AddressObjektes zuständig, dabei wird es entweder mit den richtigen Daten gefüllt oder deserialisiert.

   Address: Diese Klasse beinhaltet das serialisierte Objekt Address, das zu validieren ist. Dabei wir das Interface serialized implementiert.

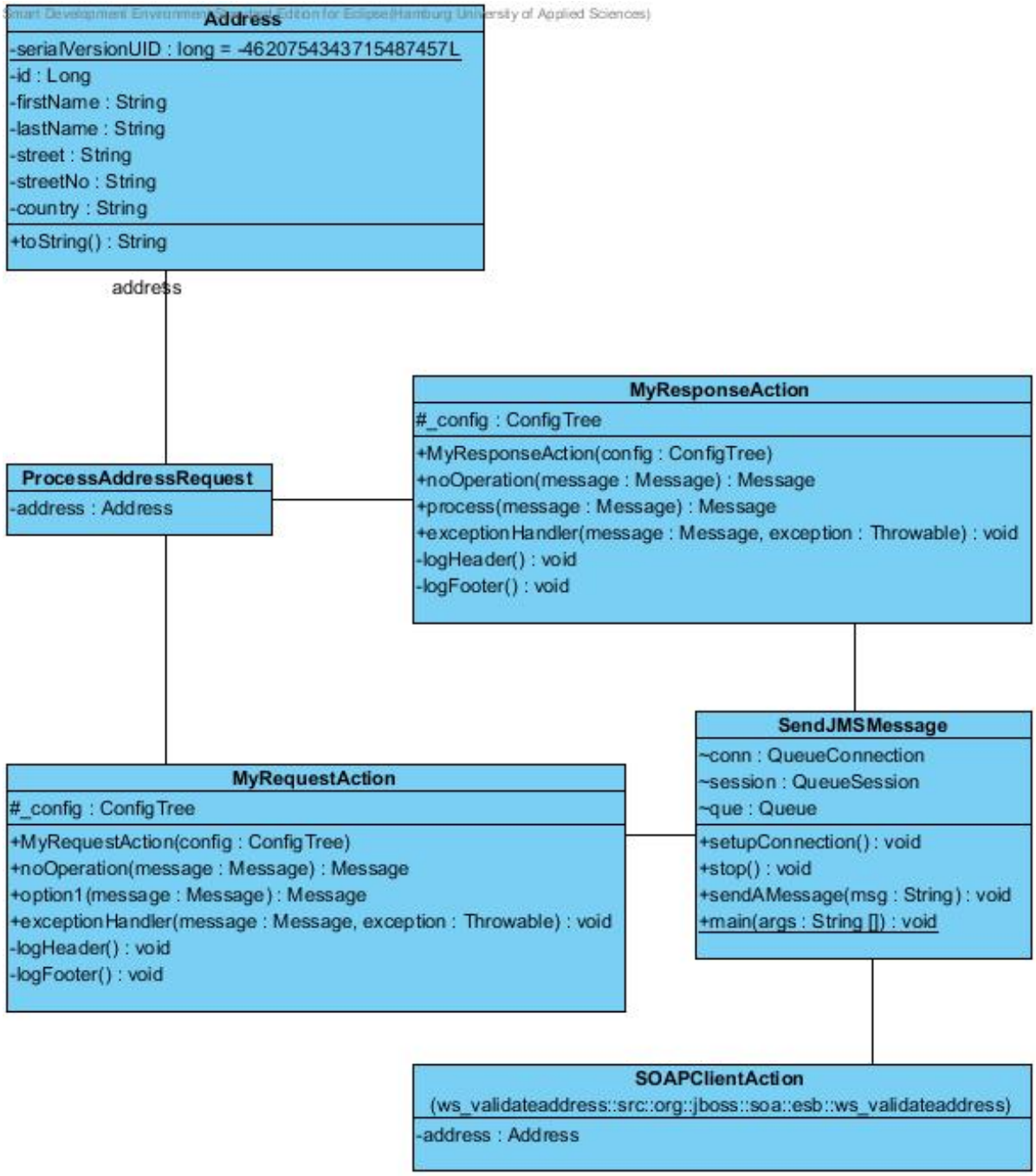

Abbildung 5.9: Client-Anwendung des AddressValidator

5.2.4 Zusammenfassung

In diesem Abschnitt wurde das Design der neuen Architektur vorgestellt. Dabei wurde die Zielarchitektur dargestellt und die Systemanforderungen wurden festgelegt. Nach der Dedinition der Systemanforderungen wurde der JBoss ESB vorgestellt, um das Design für die Kommunikation zwischen den einzelnen Komponenten und dem JBoss zu definieren. Das Komponenten-Modell wurde vorgestellt und in Form eines Klassendiagramms verdeutlicht. Beim Entwurf der Klassendiagramme wurde gezeigt, wie sich diese in der Softwarearchitektur der Zielplattform integrieren.

Durch das Re-Engineering des Systems wird die Wiederverwendbarkeit vorhandener Komponenten und die Erweiterbarkeit des Systems mit neuen Komponenten vereinfacht. Dazu wurde ein Entwurf für die Businesslogik erstellt, der die Wiederverwendbarkeit durch Nutzung von Factories erfüllt und die Erweiterbarkeitkeit durch Verwendung von Interfaces und durch Auslagern der Adressen-Transformationen in den zuständigen Klassen ermöglicht. Zuletzt wurde die Architektur einer prototypischen Anwendung vorgestellt.

Im nächsten Abschnitt findet die Realisierung des Prototyps statt, um das Designkonzept zu evaluieren. Dabei werden die technischen Randbedingungen definiert und die benötigten Tools vorgestellt, die für die Realisierung des Prototyps benötigt werden. Eine geeignete technische Plattform ist durch die Verwendung des JBoss ESB erfüllt.

6 Prototypische Evaluierung

Nachdem das Design und die Klassendiagramme der neuen Komponenten im vorigen Abschnitt vorgestellt wurden, findet in diesem Abschnitt eine prototypische Realisierung statt, um das Designkonzept zu evaluieren. Für die Implementierung und die Integration des AddressValidator in die neue Architektur mittels ein JBoss ESB werden zwei Schnittstellen benötigt. Die Schnittstelle Shop-System <-> JBoss ESB und die Schnittstelle JBoss-ESB <->AddressValidator. Dabei werden JBoss-Messages zwischen den Anwendungen ausgetauscht. Bei der prototypischen Evaluierung wird eine kleine Anwendung implementiert, die ein Message Objekt anlegt, mit Informationen befüllt und an den JBoss ESB verschickt. Das JBoss ESB soll diese Message empfangen, bearbeiten und ausgeben. Zuerst werden die Tools und Softwares vorgestellt, die für die Realisierung der Fallstudie benötigt werden. Im Anschluss findet die Implementierung der Beispielanwendung statt.

6.1 Entwicklungsumgebung und benötigte Tools

Bei einer Integrationsaufgabe werden immer zwei Systeme angefasst. Im Rahmen dieser Arbeit werden Entwicklungen auf dem existierenden Shop-System durchgeführt. Dabei wird die Komponenten-Logik aus dem Shop-System herausgenommen und stattdessen werden entfernte Aufrufe der ausgelagerten Komponenten aufgerufen. Auch die neue Komponente wird entwickelt und als Service zur Verfügung gestellt. Die Shop-System-Entwicklungen werden auf einer speziellen Entwicklungsplattform durchgeführt (Enfinity Studio).

Die Komponenten-Entwicklungen finden auf dem JBoss ESB statt, dabei wird die Entwicklungsumgebung Eclipse verwendet. Im Folgenden werden beide Entwicklungsumgebungen vorgestellt.

6.1.1 Entwicklungsplattform (Enfinity Studio)

Enfinity Studio ist das Werkzeug, mit dem sich der gesamte Entwicklungsprozess für die E-Commerce-Applikation Intershop Enfinity Suite 6 abbilden lässt. Enfinity Studio basiert auf

dem Eclipse Projekt sowie Intershop Enfinity Suite 6-spezifischen Eclipse Plug-Ins. Die Gesamtmenge dieser Plug-Ins unterstützt den Anwender in den folgenden Stufen der Enfinity Suite 6-basierten Entwicklung:

- Mit der graphischen objektorientierten Modellierung bietet Enfinity Studio die Möglichkeit an, die vorhandene Business-Logik der Enfinity Suite 6 visuell zu verändern (visuelles Programmieren).
- Automatische Source Code-Generierung für die modellierten Klassen der objektrelationalen Schicht
- Verwaltung des gesamten Java-Quellcodes auf Basis des komponentenorientierten Enfinity Cartridge-Modells
- Erzeugung und Bearbeitung von Pipelines, welche das graphische Ausdrucksmittel für die Modellierung von Geschäftsprozessen innerhalb von Enfinity Suite 6 sind.
- Unterstützung bei der Entwicklung der Web-basierten Benutzeroberfläche von Enfinity Suite
- Unterstützung bei der Entwicklung von Webservices (Consumer und Provider)
- Unterstützung bei der Template-Entwicklung.

6.1.2 JBoss ESB Applikationsserver

Als Laufzeitumgebung für Java basierte Webapplikationen muss ein Javafähiger Applikationsserver verwendet werden. Im Rahmen dieser Arbeit wird der Opensource Applikationsserver JBoss benutzt. Auch JBoss bietet eine eigene Eclipse Distribution unter dem Namen JBoss IDE an. Diese enthält neben der Steuerung des Servers über die IDE noch die Entwicklung von anderen JBoss Projekten, wie etwa jBPM. Im Fall, dass die JBoss IDE verwendet wird, müssen die oAW Plug-Ins sowie die restlichen Plug-Ins, von denen oAW abhängig ist, nachinstalliert werden.

6.2 Realisierung

Im Design-Kapitel wurde ein Entwurf der AddressValidator Komponente erstellt. In diesem Abschnitt wird eine prototypische Anwendung entwickelt, um die theoretischen Ergebnisse auf Praxistauglichkeit zu untersuchen. Dabei wird ein JBoss Service implementiert, der eine Testaktion ausführen soll. Die Testaktion soll ein Messageobjekt synchron empfangen und ausgeben. Als Rückgabewert gibt die Testaktion ein anderes Objekt zurück, das von der Clientanwendung ausgegeben werden soll.

6.2.1 JBoss Aktion Implementierung

In diesem Abschnitt wird die JBoss Testaktion implementiert. Sie empfängt ein Message Objekt und gibt die empfangenen Informationen in Form eines strings aus, dazu wird ein neues Objekt angelegt, mit Testdaten befüllt und an die Clientanwendung zurückgegeben.

Die DataContainer bekommte folgende Attribute:

- String testName;
- int testInt;
- double testDouble;
- boolean testBoolean;
- long testLong;

Für die vorgestellten Attribute werden Getter- und Setter-Methoden erstellt. Dazu soll der DataContainer die Serialized Methode implementieren, damit eine Serialisierung der Instanzen erfolgen kann. Die Serialisierung ist vom JBoss ESB vorausgesetzt, um die Objekte in eine „serielle Form“ einer Folge von Bytes zu bringen und um die Übertragung zu ermöglichen.

Nach der Implementierung der DataCountainer wird jetzt die Testaktion implementiert:

Zuerst soll die Testaktion die Klasse AbstractActionPipelineProcessor überschreiben:

```
public class TestAction extends AbstractActionPipelineProcessor {
  public void initialise() throws ActionLifecycleException {
     // Initialize resources...
     }
  public Message process(final Message msg) throws
  ActionProcessingException {
   //Nachricht Inhalt ermitteln
   Body mb=(MapBody)msg.getBody().get();
   // Inhalt ausgeben
   if (mb.get() instanceOf(DataContainer)){
   DataContainer dc=(DataContainer)mb.get();
System.out.print(dc.toString());
   }else{
   System.out.print("'Ungültiges Objekt! ClassCast Exception"');
   }

   //neues Messageobjekt anlegen:
   Message returnMsg = MessageFactory.getInstance().getMessage();
```

```
   DataContainer data = new DataContainer();
   data.setTestString("`testString from DataContainer"');
   data.setTestInt(13);
   data.setTestBoolean(true);
   returnMsg.getBody().put(data);
return returnMsg;
     }
  public void destroy() throws ActionLifecycleException {
        // Cleanup resources...
     }
}
```

Defaultmäßig wird die Methode process ausgeführt, wenn keine andere Methode definiert ist. Die Geschäftslogik kann innerhalb dieser Aktion implementiert werden. In der prototypischen Realisierung wird keine Geschäftslogik implementiert sondern ein TestObjekt (DataContainer) empfangen und ausgegeben. Dazu wird eine neue Instance des Message-Objekts angelegt, mit Informationen befüllt und zurückgegeben.

Als Rückgabewert des Methode process wird ein Message-Objekt zurückgegeben, weil die Kommunikation innerhalb der JBoss ESB ausschließlich über Messages durchgeführt wird.

6.2.2 Serviceimplementierung

In diesem Abschnitt wird der JBoss Service definiert. Dabei soll er die implementierte Testaktion (Abschnitt 6.2.1) ausführen.

Die Services im JBoss ESB werden in der JBoss-esb.xml Datei folgendermaßen definiert:

```
<?xml version = "`1.0"' encoding = "`UTF-8"'?>
<jbossesb xmlns="'http://anonsvn.labs.jboss.com/labs/jbossesb/
trunk/schemas/xml/jbossesb-1.0.1.xsd"' invmScope="'GLOBAL"'>
<services>
<service category="'TestCategory"' name="'TestService"' >
<actions>

<action name="action1" class="com.jboss.elayadi.TestAction">
</action>
</actions>
</service>
</services>
</jbossesb>
```

In der JBoss-esb.xml Datei wurde der JBoss Service definiert. Er wird einer Categorie mit dem Namen „TestCategory“ zugewiesen und bekommt den Namen „TestService“. Sobald der Service aufgerufen wird, wird die TestAktion automatisch ausgeführt. Der Aufruf des Service wird bei dieser prototypischen Realisierung synchron durchgeführt.

6.2.3 Client Anwendung

In diesem Abschnitt wird die Clientanwendung implementiert, die den JBoss Service auffindet und aufruft. Dazu wird in der Clientanwendung das Messageobjekt angelegt und mit Testdaten befüllt. Nach dem Aufruf des JBoss Service kommt ein Message-Objekt zurück, das ausgewertet und ausgegeben werden soll.

Die Client Anwendung kann den JBoss-Service auf folgende Weise finden und aufrufen:

```
ServiceInvoker invoker =
new ServiceInvoker("'TestCategory"', "'TestService"');
```

Der ServiceInvoker sucht den Service in der Service-Repository durch den Servicenamen und die ServiceCategory.

Der Aufruf der TestService wird synchron auf folgender Weise durchgeführt, damit ein Ergebniss zurückgeliefert werden kann:

```
ServiceInvoker invoker =
new ServiceInvoker("'SOAPClients"', "'TestSOAPClient"');

   //neues Messageobjekt anlegen:
   Message initialMsg = MessageFactory.getInstance().getMessage();
   DataContainer data = new DataContainer();
   data.setTestString("'testString from DataContainer"');
   data.setTestInt(13);
   data.setTestBoolean(true);
   initialMsg.getBody().put(data);
   Message responce= invoker.deliverSync(initialMsg,5000l);
   Body mb=(MapBody)responce.getBody().get();
   // Inhalt ausgeben
   if (mb.get() instanceOf(DataContainer)){
   DataContainer dc=(DataContainer)mb.get();
System.out.print(dc.toString());
```

```
}else{
System.out.print("'Ungültiges Objekt! ClassCast Exception"');
}
```

Zuerst wird der Service unter CategoryName und den ServiceNamen gesucht, danach wird eine Instanz des Message Objekts angelegt. Dazu wird ein DataContainer angelegt und mit Daten befüllt. Der DataContainer wird dem MessageObjekt angehängt. Der Service wird dann durch den Aufruf der Methode invoker.deliverSync(initialMsg,5000l) ausgeführt. Dabei wird die TestAction (Abschnitt 6.2.1) ausgerufen. Der Client bekommt dann ein Responceobjekt und gibt es aus. Zur Sicherheit und um ein ClassCastException zu vermeiden, wird der Inhalt des ResponceBody darauf geprüft, ob das eine Instance des DataContainer Objekts ist. Erst nach erfolgreicher Prüfung wird das ResponceBody Objekt convertiert.

7 Fazit

Dieses Kapitel ist in zwei Abschnitte unterteilt. Im ersten Abschnitt erfolgt eine Zusammenfassung der Ziele sowie der Ergebnisse dieser Arbeit.

Der zweite Abschnitt besteht aus einem Ausblick auf die Erweiterungsmöglichkeiten der entworfenen SOA.

7.1 Zusammenfassung der Ergebnisse

Die sich kontinuierlich verändernden Anforderungen an ein E-Commerce System stellen hohe Anforderungen an die Systemarchitektur. Häufige kurzfristige Änderungen verlangen eine Architektur, welche die Entwickler dabei unterstützt, schnell auf Anforderungen reagieren zu können. Durch diese Anpassungen und die kontinuierliche Weiterentwicklung eines E- Commerce-Systems besteht immer die Gefahr, dass eine Integrationsproblematik entsteht und somit die Kosten und Komplexität der IT-Projekte wachsen. In Rahmen dieser Arbeit wurde das Designkonzept einer neuen Architektur auf Basis von SOA entwickelt, das eine solche Integrationsproblematik in Zukunft vermeiden soll.

Es wurde eine SOA unter Zuhilfenahme eines JBoss ESB realisiert. Durch einen Re-Engineering-Prozess konnte der AdressValidator als Service extrahiert und mittels des ESB integriert werden. Das entwickelte Designkonzept wurde im Weiteren evaluiert und als tragfähig bewertet.

SOA als Architekturstil hat im Rahmen dieser Arbeit viele Vorteile aufzeigen können.

- Erweiterbarkeit - Die Architektur ist durch die Auslagerung einzelner Services hinter generischen Interfaces leichter erweiterbar.
- Wiederverwendbarkeit - Durch die Einbindung von Komponenten als Services lassen sich diese leicht in anderen Teilen der Gesamtarchitektur wieder verwenden.
- Wartbarkeit - Die einzelnen Komponenten lassen sich durch die Kapselung als Service hinter dem WrapperInterface und die damit verbundene Trennung von dem Gesamtsystem leichter weiterentwickeln, testen und auch austauschen. Ob dies den Wartungsaufwand in der weiteren Entwicklung wirklich verringert, wurde im Rahmen dieser Arbeit nicht weiter untersucht.
- Betriebssicherheit - Durch die Verwendung des JBoss ESB lassen sich Services leicht redundant und auch in verschiedenen Realisierungsformen betreiben und je nach Verfügbarkeit einbinden.
- Erweiterbarkeit - Durch die Verwendung geeigneter Design Patterns (z. B. Facade Gamma u. a. (1995)) und die dadurch resultierende generische Anbindung von Services, als auch durch die einfache Anbindung als Service, lässt sich das E-Commerce System gut erweitern.

Dem entgegenzusetzen sind die für die Migration erforderlichen Aufwände sowie anfänglichen Schwierigkeiten mit einer zusätzlichen Abhängigkeit, dem Softwaredesign der Businesslogik.

Die Integration von bestehenden Funktionalitäten als Service kann hierbei entweder als Auf- ruf der bestehenden Komponente und deren Einbindung in die SOA geschehen, als Neu- entwicklung oder als teilweiseNeuentwicklung. Durch die teilweiseNeuentwicklung ist der neue Ser-

vice besser wartbar und kann sich besser in die Servicelandschaft integrieren. Beim Wrapping-Ansatz wird das gesamte wertvolle Wissen aus dem Legacy-System übernommen.

Neue Vertriebs-Kanäle und Mobile Clients können mit geringerem Aufwand und wenig Anpassungen im Shop-System integriert werden. Das Shop-System selbst kann ausgetauscht werden und die implementierten Funktionalitäten gehen nicht verloren und können vom neuen Shop-System übernommen werden.

Die Ergebnisse der Untersuchungen und die prototypische Evaluierung durch die Beispielanwendung zeigen, dass das Design tragfähig ist. Die Verwendung des ESB bei der Service-Orientierung bildet ein wichtiges technisches Hilfsmittel ab und spielt eine zentrale Rolle. Diese Arbeit hat gezeigt und bestätigt, dass eine nachträgliche Anwendung von SOA in einem Legacy E-Commerce-System möglich ist.

Als Fazit muss festgestellt werden, dass SOA als Architektur nicht automatisch eine leichter wartbare und erweiterbare Software hervorbringt. Auch die Aufwände für die Migration sowie die anfängliche Einarbeitung sind nicht zu vernachlässigen.

Das Softwaredesign der Businesslogik im Rahmen einer technischen Architektur spielt eine große Rolle bei der Wartbarkeit und Erweiterbarkeit der Architektur.

7.2 Ausblick

Eine komplette Einführung von SOA und die Anwendung des Designkonzeptes lässt sich teilweise mit etwas geringerer Komplexität realisieren. Dabei soll eine komplette Analyse des Altsystems durchgeführt werden und die Funktionalitäten sollen ermittelt werden, die mehrfach und von unterschiedlichen Anwendungen verwendet werden können. Danach können diese Funktionalitäten extrahiert werden. Durch das Auslagern von Funktionalitäten wird eine Migration der Softwarekomponente erforderlich. Für diese Migration wird der Cold-Turkey-Ansatz bevorzugt.

Bei der Einführung der SOA werden die Funktionalitäten einzeln ausgelagert und als Services angeboten. Das spricht auch dafür, dass der Cold-Turkey-Ansatz (3.1.2) verwendet wird, da dieser eine in mehreren Schritten geplante Datenmigration ermöglicht. Außerdem ist der Cold-Turkey-Ansatz am besten geeignet für kleine Migrationsprozesse.

Mit dem etabliertem Konzept kann im nächsten Schritt überlegt werden, ob noch zusätzliche Komponenten aus dem Kern des Shop-Systems extrahiert werden können. Dann wären, als zweiter Schritt, diese Kernfunktionalitäten als Extrakomponenten anzupassen. Beim AdressValidator ging es um die Entscheidung, den Wrapping-Ansatz (Abschnitt 3.1.1) zu verwenden, um die Funktionalität in die neuen Komponenten auszulagern. Für ähnliche Funktionalitäten der AdressValidator kann die gleiche Softwarearchitektur verwendet werden. Unter Anwendung etwas anderer Design Pattern im Rahmen der angegebenen Softwarearchitektur, ist es möglich, weitere Komponenten zu extrahieren und eine schrittweise Migration nach Cold-Turkey-Ansatz durchzuführen. Folgende Funktionalitäten können mit dem gleichen Verfahren für Adressvalidierung aus dem Shop-System als extra Komponente separiert werden:

- Bonitätsprüfung mit CreditReform Re-Engineering, auslagern und ggf. mit zusätzlichen Bonitätsdiensten erweitern.
- Produktsuche in einer Extra Module anbieten.
- BestellungsManager implementieren.

Beim Re-Engineering-Prozess wird dafür gesorgt, das Problem zu lösen, bestehende Komponenten herauszulösen, sodass sie möglichst erweiterbar sind. Bei einem parallelen Re-Engineering von Funktionalitäten könnte dieser Prozess aufwändiger und komplexer werden, dabei wird ein weiterer Ansatz empfohlen, um Duplikate und blackboxes zu vermeiden und um eventuelle zukünftige Wartungen zu vereinfachen. Dieser Ansatz ist die Neuentwicklung von Teilkomponenten (Abschnitt 3.1.1). Dabei wird das gesamte System in Komponenten zerlegt und dieser Ansatz entscheidet, welche davon eine Neuentwicklung benötigen und bei welchen ein Wrapping-Ansatz ausreichend ist.

Nach dem Auslagern der Funktionalitäten wird das Shop-System durchsucht und die Methoden aufrufe werden durch entfernte Aufrufe ersetzt. Danach findet eine Bereinigung des source-codes im Shop-System statt.

Die Ergebnisse dieser Arbeit haben bestätigt, dass eine gute Technologie alleine nicht ausreichend ist, um eine gute Software zu entwickeln, bzw. zu erzielen. Viel wichtiger ist das Design der Software in Abstimmung mit dem verwendeten Architekturkonzept. Es geht nicht darum, eine tolle Architektur zu verwenden, die einen coolen Namen hat. Es geht darum, einen fundierten Software-Entwurf zu haben, den man mit einer guten Technologie umsetzen kann.

Literaturverzeichnis

[Baghdadi 2005] BAGHDADI, Youcef: A web services-based business interactions manager to support electronic commerce applications. In: *ICEC '05: Proceedings of the 7th international conference on Electronic commerce*. New York, NY, USA : ACM, 2005, S. 435–445. – ISBN 1-59593-112-0

[BIEN 2006] BIEN, Adam: Enterprise Architekturen - Leifaden für effiziente Software-Entwicklung. In: *Enterprise Architekturen - Leifaden für effiziente Software-Entwicklung*, O'Reilly, 2006. – ISBN 3-935042-99-4

[Bisbal u. a. 1999] BISBAL, J. ; LAWLESS, D. ; WU, Bing ; GRIMSON: Legacy Information Systems: Issues and Directions. In: *Software Engineering Conference, 1999. Asia Pacific ... and International Computer Science Conference 1997. APSEC '99 and ICSC '99. Proceedings*, 2-5 1999, S. 529 –530

[Bisbal u. a. 1997] BISBAL, J. ; LAWLESS, D. ; WU, Bing ; GRIMSON, J. ; WADE, V. ; RICHARDSON, R. ; O'SULLIVAN, D.: An overview of legacy information system migration. In: *Software Engineering Conference, 1997. Asia Pacific ... and International Computer Science Conference 1997. APSEC '97 and ICSC '97. Proceedings*, 2-5 1997, S. 529 –530

[Bisbal und Cheng 2004] BISBAL, Jesús ; CHENG, Betty H. C.: Resource-based approach to feature interaction in adaptive software. In: *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*. New York, NY, USA : ACM, 2004, S. 23–27. – ISBN 1-58113-989-6

[Bundesverband Informationswirtschaft 2010] BUNDESVERBAND INFORMATIONSWIRTSCHAFT, Telekommunikation und neue Medien e.: Bundesverband Informationswirtschaft, Telekommunikation und neue Medien e.V. In: *Enterprise Service Bus*, O'Reilly Media, Inc., 2010

[CHAPPEL 2003] CHAPPEL, Tyler;: Java Web Services. In: *Java Web Services*, O'Reilly, 2003. – ISBN 3-89721-284-6

[David A Chappell; HUMM 2004] DAVID A CHAPPELL; HUMM, Bernhard: Enterprise Service Bus. In: *Enterprise Service Bus*, O'Reilly Media, Inc., 2004. – ISBN 978-0-596-00675-4

[Erl. 2005a] ERL., Thomas: Revisiting the definitive SOA definition. In: *searchwebservices* http://searchwebservices.techtarget.com/originalContent (2005)

[Erl. 2005b] ERL., Thomas: Service-Oriented Architecture: Concepts, Technology, and Design. In: *Prentice Hall* (2005)

[Feldhorst u. a. 2009] FELDHORST, S. ; LIBERT, S. ; HOMPEL, M. ten ; KRUMM, H.: Integration of a legacy automation system into a SOA for devices. In: *Emerging Technologies Factory Automation, 2009. ETFA 2009. IEEE Conference on*, 22-25 2009, S. 1 –8. – ISSN 1946-0759

[Gamma u. a. 1995] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: Design Patterns. Reading, MA : Addison Wesley, 1995

[Gerardo Canfora 2006] GERARDO CANFORA, Porfirio T.: Migrating Interactive Legacy Systems To Web Services. In: *Migrating Interactive Legacy Systems To Web Services*, 2-5 2006, S. 312 –320

[Grace Lewis 2005] GRACE LEWIS, Dennis S.: Service-Oriented Migration and Reuse Technique (SMART). In: *Prentice Hall* http://ieeexplore.ieee.org.globalproxy.cvt.dk/iel5/11138/35653/01691651.pdf (2005)

[HESS 2005] HESS, Bernhard: Regeln für serviceorientierte Architekturen hoher Qualität; Informatik Spektrum Band. In: *Regeln für serviceorientierte Architekturen hoher Qualität*, Springer-Verlag, 2005, S. 395–411. – ISBN 978-0-7695-3650-7

[Li 2010] LI, Baoan: Research and Application of SOA Standards in the Integration on Web Services. In: *Education Technology and Computer Science (ETCS), 2010 Second International Workshop on* Bd. 2, 6-7 2010, S. 492 –495

[LIEBHART 2007] LIEBHART, Daniel: SOA goes real. In: *SOA goes real*, Springer-Verlag, 2007. – ISBN 3-446- 41088-0

[Ltd 2007] LTD, Quocirca: Functional Re-use and SOA The Service Interface as an Enabler for Software Re-use Best Practice. In: *Prentice Hall* (2007)

[Mahmoud und Gomez 2008] MAHMOUD, T. ; GOMEZ, J.M.: Integration of Semantic Web Services Principles in SOA to Solve EAI and ERP Scenarios; Towards Semantic Service Oriented Architecture. In: *Information and Communication Technologies: From Theory to Applications, 2008. ICTTA 2008. 3rd International Conference on*, 7-11 2008, S. 1 –6

[McKendrick 2007] MCKENDRICK, Joe: SOA business case: smaller is better. In: *Prentice Hall* (2007)

[Mughrabi 2007] MUGHRABI, Hanafi: A technical Investigation of SOA, and Ecommerce Systems. In: *Prentice Hall* (2007)

[Papazoglou und Heuvel 2007a] PAPAZOGLOU, Mike P. ; HEUVEL, Willem-Jan: Service oriented architectures: approaches, technologies and research issues. In: *The VLDB Journal* 16 (2007), July, S. 389–415. – URL http://dx.doi.org/10.1007/s00778-007-0044-3. – ISSN 1066-8888

[Papazoglou und Heuvel 2007b] PAPAZOGLOU, Mike P. ; HEUVEL, Willem-Jan: Service oriented architectures: approaches, technologies and research issues. In: *The VLDB Journal* 16 (2007), Nr. 3, S. 389–415. – ISSN 1066-8888

[Schoop 2003] SCHOOP, M.: Business Communication in Electronic Commerce. In: *Business Communication in Electronic Commerce*. Communications of the ACM : RWTH Aachen, 2003, S. 631–636. – ISBN 978-0-7695-3650-7

[Shan 2004] SHAN, T.C.: Building a service-oriented ebanking platform. In: *Services Computing, 2004. (SCC 2004). Proceedings. 2004 IEEE International Conference on*, 15-18 2004, S. 237 – 244

[T. Malone 2009] T. MALONE, J. Yates und R. B.: Eletronic Markets and Eletronic Hierarchies. In: *Eletronic Markets and Eletronic Hierarchies*. Communications of the ACM : Communications of the ACM, 2009, S. 631–636. – ISBN 978-0-7695-3650-7

[Wang u. a. 2009a] WANG, Jun ; YU, AiRong ; ZHANG, XiaoYi ; QU, Lei: A dynamic data integration model based on SOA. In: *Computing, Communication, Control, and Management, 2009. CCCM 2009. ISECS International Colloquium on* Bd. 2, 8-9 2009, S. 196 –199

[Wang u. a. 2009b] WANG, Zhongjie ; XU, Xiaofei ; CHU, Dianhui ; MA, Chao: A Case Study on Bi-lateral Resource Integration Oriented Marine Logistics Service System. In: *SCC '09: Proceedings of the 2009 IEEE International Conference on Services Computing*. Washington, DC, USA : IEEE Computer Society, 2009, S. 458–465. – ISBN 978-0-7695-3811-2

[Wu u. a. 1999a] WU, Bing ; LAWLESS, D. ; BISBAL, J. ; GRIMSON: A Brief Review of Problems, Solutions and Research Issues. In: *Software Engineering Conference, 1999. Asia Pacific ... and International Computer Science Conference 1997. APSEC '99 and ICSC '99. Proceedings*, 2-5 1999, S. 312 –320

[Wu u. a. 1999b] WU, Bing ; LAWLESS, D. ; BISBAL, J. ; GRIMSON, J. ; WADE, V. ; O'SULLIVAN, D. ; RICHARDSON, R.: Legacy systems migration-a method and its tool-kit framework. In: *Software Engineering Conference, 1997. Asia Pacific ... and International Computer Science Conference 1997. APSEC '97 and ICSC '97. Proceedings*, 2-5 1999, S. 312 –320

[Wu u. a. 1997] WU, Bing ; LAWLESS, D. ; BISBAL, J. ; RICHARDSON, R. ; GRIMSON, J. ; WADE, V. ; O'SULLIVAN, D.: The Butterfly Methodology: a gateway-free approach for migrating legacy information systems. In: *Engineering of Complex Computer Systems, 1997. Proceedings., Third IEEE International Conference on*, 8-12 1997, S. 200 –205

[Xiaoqing und Hao 2008] XIAOQING, Huang ; HAO, Jiang: SOA-based integration of electric utility in open electric market. In: *Electric Utility Deregulation and Restructuring and Power Technologies, 2008. DRPT 2008. Third International Conference on*, 6-9 2008, S. 2245 – 2250

[Yee u. a. 2009] YEE, Keng Y. ; TIONG, Ang W. ; TSAI, Flora S. ; KANAGASABAI, Rajaraman: OntoMobiLe: A Generic Ontology-Centric Service-Oriented Architecture for Mobile Learning. In: *MDM '09: Proceedings of the 2009 Tenth International Conference on Mobile Data Management: Systems, Services and Middleware*. Washington, DC, USA : IEEE Computer Society, 2009, S. 631–636. – ISBN 978-0-7695-3650-7

[Zhao u. a. 2008] ZHAO, Yuanping ; WU, Jun ; SHU, Huaying: The Fractal Management of SOA-Based Services Integration. In: *Information Management, Innovation Management and Industrial Engineering, 2008. ICIII '08. International Conference on* Bd. 3, 19-21 2008, S. 420 –424

[Zhuopeng Zhang 2004] ZHUOPENG ZHANG, Hongji Y.: Incubating Services in Legacy Systems for Architectural Migration. In: *ieeexplore* http://ieeexplore.ieee.org.globalproxy.cvt.dk/iel5/9444/29994/01371920.pdf (2004)

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 23. Dezember 2010
Ort, Datum

Unterschrift